



Reader Protocol Standard, Version 1.1

Ratified Standard

June 21, 2006

Disclaimer

EPCglobal IncTM is providing this document as a service to interested industries. This document was developed through a consensus process of interested parties.

Although efforts have been to assure that the document is correct, reliable, and technically accurate, EPCglobal Inc makes NO WARRANTY, EXPRESS OR IMPLIED, THAT THIS DOCUMENT IS CORRECT, WILL NOT REQUIRE MODIFICATION AS EXPERIENCE AND TECHNOLOGICAL ADVANCES DICTATE, OR WILL BE SUITABLE FOR ANY PURPOSE OR WORKABLE IN ANY APPLICATION, OR OTHERWISE. Use of this document is with the understanding that EPCglobal Inc has no liability for any claim to the contrary, or for any damage or loss of any kind or nature.

Copyright notice

© 2006, EPCglobal Inc.

All rights reserved. Unauthorized reproduction, modification, and/or use of this document is not permitted. Requests for permission to reproduce should be addressed to epcglobal@epcglobalinc.org.

EPCglobal Inc.TM is providing this document as a service to interested industries. This document was developed through a consensus process of interested parties. Although efforts have been to assure that the document is correct, reliable, and technically accurate, EPCglobal Inc. makes NO WARRANTY, EXPRESS OR IMPLIED, THAT THIS DOCUMENT IS CORRECT, WILL NOT REQUIRE MODIFICATION AS EXPERIENCE AND TECHNOLOGICAL ADVANCES DICTATE, OR WILL BE SUITABLE FOR ANY PURPOSE OR WORKABLE IN ANY APPLICATION, OR OTHERWISE. Use of this Document is with the understanding that EPCglobal Inc. has no liability for any claim to the contrary, or for any damage or loss of any kind or nature

Abstract

This document defines Version 1.1 of the protocol by which tag readers interact with EPCglobal compliant software applications. The terms “tag reader” and “reader” include RFID tag readers, supporting any combination of RF protocols, fixed and hand-held, etc. It also includes readers of other kinds of tags, such as bar codes. Tag readers, despite the name, can also have the ability to write data into tags.

Audience for this document

The target audience for this specification includes:

- EPC Middleware vendors
- Reader vendors
- Application developers
- System integrators

Status of this document

This section describes the status of this document at the time of its publication. Other documents MAY supersede this document. The latest status of this document series is maintained at EPCglobal. See www.epcglobalinc.org for more information.

This draft is an EPCglobal **Recommended Specification** for review by members of the EPCglobal Board of Governors.

Comments on this document should be sent to the EPCglobal Software Action Group Reader Protocol Working Group mailing list
sag_reader_protocol@epclinklist.epcglobalinc.org.

Date of Change	Version	Reason for Change	Summary of Change
10/14/05	1.1 Candidate	SAG Vote as required by SDP	LCWD voted by SAG and Became a Candidate Specification
2/2/06	1.1 Proposed	Prototype Testing as required by SDP	Small informative changes to specification made as a result of prototype testing resulted in advancing it to a Proposed Specification
4/10/06	1.1 Recommended	Completed review by TSC and BSC	TSC review completed 3/27/06; BSC review completed 4/10/06
6/21/06	1.1 Ratified	EPCglobal Board	

62

63

64 **Statement of Conformance**

65

66 Software or devices incorporating this specification shall not claim conformance unless
67 they comply with the following:

- 68 • All mandatory clauses [SHALLs] in this specification (except those marked as
69 optional),
- 70 • The Reader Protocol Conformance Requirements (separate document)

71 Note that Conformance may also require a license from the owner of any intellectual
72 property utilized to the extent of necessary claims made by said device or software. Use
73 of any XML schemas may not be in compliance with applications covered by state or
74 federal laws.

75

Table of Contents

76		
77	Reader Protocol 1.1.....	1
78	Proposed Specification	1
79	March 21, 2006.....	1
80	Table of Contents.....	4
81	List of Figures	14
82	List of Tables	15
83	List of Tables	15
84	1 Introduction	17
85	2 Terminology and Typographical Conventions.....	17
86	3 Protocol Layers	18
87	3.1 Message Channels	18
88	4 Reader Layer – Conceptual Overview	19
89	4.1 Read Subsystem.....	21
90	4.1.1 Sources.....	22
91	4.1.2 ReadTrigger	22
92	4.1.3 Data Acquisition Stage	22
93	4.1.3.1 Tag RF Protocol Binding	24
94	4.1.4 Read Filter Stage.....	24
95	4.2 Event Subsystem	24
96	4.2.1 Smoothing/Event Generation Stage.....	24
97	4.3 Output Subsystem.....	27
98	4.3.1 Data Selector Stage	28
99	4.3.2 Report Buffer Stage	28
100	4.3.3 Notification Trigger	28
101	4.4 Communication Subsystem	28
102	4.4.1 Message/Transport Binding Stage	28
103	5 Object Model.....	30
104	5.1 ReaderDevice Object.....	32
105	5.2 Sources.....	33
106	5.2.1 Source Organization.....	34

107	5.3	Triggers.....	37
108	5.4	TagSelectors	39
109	5.4.1	Multiple Tag Selectors.....	40
110	5.5	Events	40
111	5.6	Channels	41
112	5.7	DataSelectors	41
113	5.8	TagFields and TagFieldValues.....	43
114	5.9	Object Names	43
115	6	Reader Layer – Commands	44
116	6.1	Object ReaderDevice	47
117	6.1.1	ReaderDevice.getEPC.....	47
118	6.1.2	ReaderDevice.getManufacturer	47
119	6.1.3	ReaderDevice.getModel.....	48
120	6.1.4	ReaderDevice.getHandle	48
121	6.1.5	ReaderDevice.setHandle.....	49
122	6.1.6	ReaderDevice.getName	49
123	6.1.7	ReaderDevice.setName.....	50
124	6.1.8	ReaderDevice.getRole	50
125	6.1.9	ReaderDevice.setRole	51
126	6.1.10	ReaderDevice.getTimeTicks	52
127	6.1.11	ReaderDevice.getTimeUTC	52
128	6.1.12	ReaderDevice.setTimeUTC.....	53
129	6.1.13	ReaderDevice.getManufacturerDescription	53
130	6.1.14	ReaderDevice.getCurrentSource	54
131	6.1.15	ReaderDevice.setCurrentSource.....	54
132	6.1.16	ReaderDevice.getCurrentDataSelector.....	55
133	6.1.17	ReaderDevice.setCurrentDataSelector	56
134	6.1.18	ReaderDevice.removeSources	56
135	6.1.19	ReaderDevice.removeAllSources.....	57
136	6.1.20	ReaderDevice.getSource.....	57
137	6.1.21	ReaderDevice.getAllSources.....	58
138	6.1.22	ReaderDevice.removeDataSelectors	58

139	6.1.23	ReaderDevice.removeAllDataSelectors	59
140	6.1.24	ReaderDevice.getDataSelector	60
141	6.1.25	ReaderDevice.getAllDataSelectors	60
142	6.1.26	ReaderDevice.removeNotificationChannels	61
143	6.1.27	ReaderDevice.removeAllNotificationChannels	61
144	6.1.28	ReaderDevice.getNotificationChannel	62
145	6.1.29	ReaderDevice.getAllNotificationChannels	63
146	6.1.30	ReaderDevice.removeTriggers	63
147	6.1.31	ReaderDevice.removeAllTriggers	64
148	6.1.32	ReaderDevice.getTrigger	64
149	6.1.33	ReaderDevice.getAllTriggers	65
150	6.1.34	ReaderDevice.removeTagSelectors	66
151	6.1.35	ReaderDevice.removeAllTagSelectors	66
152	6.1.36	ReaderDevice.getTagSelector	67
153	6.1.37	ReaderDevice.getAllTagSelectors	67
154	6.1.38	ReaderDevice.removeTagFields	68
155	6.1.39	ReaderDevice.removeAllTagFields	69
156	6.1.40	ReaderDevice.getTagField	69
157	6.1.41	ReaderDevice.getAllTagFields	70
158	6.1.42	ReaderDevice.resetToDefaultSettings	70
159	6.1.43	ReaderDevice.reboot	71
160	6.1.44	ReaderDevice.goodbye	71
161	6.1.45	ReaderDevice.getReadPoint	72
162	6.1.46	ReaderDevice.getAllReadPoints	73
163	6.2	Object Source	73
164	6.2.1	Source.create	73
165	6.2.2	Source.getName	74
166	6.2.3	Source.isFixed	75
167	6.2.4	Source.addReadPoints	75
168	6.2.5	Source.removeReadPoints	76
169	6.2.6	Source.removeAllReadPoints	77
170	6.2.7	Source.getReadPoint	77

171	6.2.8	Source.getAllReadPoints	78
172	6.2.9	Source.addReadTriggers	78
173	6.2.10	Source.removeReadTriggers	79
174	6.2.11	Source.removeAllReadTriggers	80
175	6.2.12	Source.getReadTrigger	80
176	6.2.13	Source.getAllReadTriggers	81
177	6.2.14	Source.addTagSelectors	81
178	6.2.15	Source.removeTagSelectors	82
179	6.2.16	Source.removeAllTagSelectors	83
180	6.2.17	Source.getTagSelector	83
181	6.2.18	Source.getAllTagSelectors	84
182	6.2.19	Source.getGlimpsedTimeout	84
183	6.2.20	Source.setGlimpsedTimeout.....	85
184	6.2.21	Source.getObservedThreshold.....	85
185	6.2.22	Source.setObservedThreshold	86
186	6.2.23	Source.getObservedTimeout	86
187	6.2.24	Source.setObservedTimeout.....	87
188	6.2.25	Source.getLostTimeout.....	87
189	6.2.26	Source.setLostTimeout	88
190	6.2.27	Source.rawReadIDs	89
191	6.2.28	Source.readIDs	89
192	6.2.29	Source.read	90
193	6.2.30	Source.writeID.....	91
194	6.2.31	Source.write.....	92
195	6.2.32	Source.kill.....	93
196	6.2.33	Source.getReadCyclesPerTrigger.....	94
197	6.2.34	Source.setReadCyclesPerTrigger	95
198	6.2.35	Source.getMaxReadDutyCycle	95
199	6.2.36	Source.setMaxReadDutyCycle.....	96
200	6.2.37	Source.getReadTimeout	96
201	6.2.38	Source.setReadTimeout.....	97
202	6.2.39	Source.getSession.....	97

203	6.2.40	Source.setSession	98
204	6.2.41	Source.getMaxNumberSupported	98
205	6.3	Object TagFieldValue.....	99
206	6.3.1	TagFieldValue.TagFieldName:String.....	99
207	6.3.2	TagFieldValue.Value:binary.....	99
208	6.4	Object ReadPoint.....	99
209	6.4.1	ReadPoint.getName	99
210	6.5	Object Trigger.....	100
211	6.5.1	Trigger.create	101
212	6.5.2	Trigger.getMaxNumberSupported.....	102
213	6.5.3	Trigger.getName	103
214	6.5.4	Trigger.getType	103
215	6.5.5	Trigger.getValue	104
216	6.5.6	Trigger.fire	104
217	6.6	Object TagSelector	105
218	6.6.1	TagSelector.create.....	105
219	6.6.2	TagSelector.getMaxNumberSupported	106
220	6.6.3	TagSelector.getName.....	107
221	6.6.4	TagSelector.getTagField	107
222	6.6.5	TagSelector.getValue.....	108
223	6.6.6	TagSelector.getMask	108
224	6.6.7	TagSelector.getInclusiveFlag.....	109
225	6.7	Object CommandChannel.....	109
226	6.8	Object NotificationChannel.....	109
227	6.8.1	NotificationChannel.create	110
228	6.8.2	NotificationChannel.getName	111
229	6.8.3	NotificationChannel.getAddress.....	111
230	6.8.4	NotificationChannel.getEffectiveAddress	112
231	6.8.5	NotificationChannel.setAddress	112
232	6.8.6	NotificationChannel.getDataSelector	113
233	6.8.7	NotificationChannel.setDataSelector.....	113
234	6.8.8	NotificationChannel.addSources	114

235	6.8.9	NotificationChannel.removeSources	115
236	6.8.10	NotificationChannel.removeAllSources.....	115
237	6.8.11	NotificationChannel.getSource.....	116
238	6.8.12	NotificationChannel.getAllSources.....	116
239	6.8.13	NotificationChannel.addNotificationTriggers.....	117
240	6.8.14	NotificationChannel.removeNotificationTriggers.....	117
241	6.8.15	NotificationChannel.removeAllNotificationTriggers.....	117
242	6.8.16	NotificationChannel.getNotificationTrigger	117
243	6.8.17	NotificationChannel.getAllNotificationTriggers.....	117
244	6.8.18	NotificationChannel.readQueuedData.....	117
245	6.8.19	NotificationChannel.getMaxNumberSupported.....	117
246	6.9	Object DataSelector	117
247	6.9.1	DataSelector.create	117
248	6.9.2	DataSelector.getName	117
249	6.9.3	DataSelector.addFieldNames.....	117
250	6.9.4	DataSelector.removeFieldNames.....	117
251	6.9.5	DataSelector.removeAllFieldNames	117
252	6.9.6	DataSelector.getAllFieldNames.....	117
253	6.9.7	DataSelector.addEventFilters.....	117
254	6.9.8	DataSelector.removeEventFilters	117
255	6.9.9	DataSelector.removeAllEventFilters	117
256	6.9.10	DataSelector.getAllEventFilters	117
257	6.9.11	DataSelector.addTagFieldNames	117
258	6.9.12	DataSelector.removeTagFieldNames	117
259	6.9.13	DataSelector.removeAllTagFieldNames.....	117
260	6.9.14	DataSelector.getAllTagFieldNames	117
261	6.9.15	DataSelector.getMaxNumberSupported.....	117
262	6.10	Enumeration Objects	117
263	6.10.1	Object EventType	117
264	6.10.1.1	EventType.getSupportedTypes.....	117
265	6.10.2	Object TriggerType	117
266	6.10.2.1	TriggerType.getSupportedTypes	117

267	6.10.3	Object FieldName.....	117
268	6.10.3.1	FieldName.getSupportedNames	117
269	6.10.4	Object PredefinedTagFieldName	117
270	6.11	Object TagField.....	117
271	6.11.1	TagField.create	117
272	6.11.2	TagField.getName	117
273	6.11.3	TagField.getTagFieldName	117
274	6.11.4	TagField.setTagFieldName	117
275	6.11.5	TagField.getMemoryBank.....	117
276	6.11.6	TagField.setMemoryBank	117
277	6.11.7	TagField.getOffset.....	117
278	6.11.8	TagField.setOffset	117
279	6.11.9	TagField.getLength.....	117
280	6.11.10	TagField.setLength.....	117
281	6.11.11	TagField.getMaxNumberSupported	117
282	7	Reader Layer – Notifications	117
283	7.1	Events	117
284	7.1.1	Read Report	117
285	7.1.2	Vendor-defined Notification.....	117
286	7.2	MTB-specific notification	117
287	7.2.1	Goodbye.....	117
288	8	Error Handling.....	117
289	8.1	Error Conditions	117
290	8.2	Communication Errors	117
291	8.2.1	Communication Host-to-Reader	117
292	8.2.2	Communication Reader-to-Host	117
293	8.3	Command Errors.....	117
294	8.4	Vendor Extensions.....	117
295	9	Message/Transport Bindings (MTBs).....	117
296	9.1	Address Notation	117
297	10	Transports	117
298	10.1	Handshaking.....	117

299	10.1.1	Terminology	117
300	10.1.2	Syntax	117
301	10.1.2.1	Sender Handshake.....	117
302	10.1.2.2	Receiver Handshake.....	117
303	10.1.3	Operations.....	117
304	10.1.3.1	Initiation.....	117
305	10.1.3.2	Versioning and Capabilities Request.....	117
306	10.1.3.3	Extensibility	117
307	10.2	Serial.....	117
308	10.2.1	Transport Address Notation.....	117
309	10.2.2	Serial Interface Physical Characteristics	117
310	10.2.3	Human-to-Machine Mode (TTY).....	117
311	10.2.3.1	Connection Establishment (Connect)	117
312	10.2.3.2	Connection Termination (Disconnect).....	117
313	10.2.3.3	Message Format	117
314	10.2.4	Machine-to-Machine Mode (M2M)	117
315	10.2.4.1	Connection Establishment (Connect)	117
316	10.2.4.2	Connection Termination (Disconnect).....	117
317	10.2.4.3	Message Format	117
318	10.3	TCP.....	117
319	10.3.1	Transport Address Notation.....	117
320	10.3.2	Connection Establishment (Connect)	117
321	10.3.2.1	Control Channel	117
322	10.3.2.2	Notification Channel.....	117
323	10.3.3	Connection Termination (Disconnect)	117
324	10.3.3.1	Control Channel	117
325	10.3.3.2	Notification Channel.....	117
326	10.3.4	Message Format.....	117
327	10.4	HTTP	117
328	10.4.1	Transport Address Notation.....	117
329	10.4.2	Connection Establishment (Connect)	117
330	10.4.2.1	HTTP 1.1 is required.....	117

331	10.4.2.2	All standard RP headers named ‘RP-*’	117
332	10.4.2.3	HTTP Header ‘Content-Type’ is required	117
333	10.4.2.4	HTTP Header ‘Content-Length’ is required	117
334	10.4.2.5	Additional HTTP headers MAY be present	117
335	10.4.2.6	HTTP Request	117
336	10.4.2.7	HTTP Response	117
337	10.4.2.8	Control Channel	117
338	10.4.2.9	Notification Channel	117
339	10.4.3	Connection Termination (Disconnect)	117
340	10.4.4	Message Format	117
341	11	Message Formats	117
342	11.1	XML Message Format	117
343	11.1.1	Object Identification Scheme	117
344	11.1.2	Data Types	117
345	11.1.2.1	Enumeration Objects	117
346	11.1.3	Read Report encoding	117
347	11.1.4	Command XML message encoding (Host-To-Reader)	117
348	11.1.4.1	Message Format	117
349	11.1.4.2	Vendor Extensions	117
350	11.1.5	Reply XML Message Encoding (Reader-To-Host)	117
351	11.1.5.1	Message Format	117
352	11.1.6	Notification XML message encoding (Reader-To-Host)	117
353	11.1.6.1	ACK/NACK	117
354	11.1.7	Vendor Extensions	117
355	11.2	Text (non-XML) Message Format	117
356	11.2.1	External Encoding	117
357	11.2.2	Object Identification Scheme	117
358	11.2.3	Data Types	117
359	11.2.3.1	Enumeration Objects	117
360	11.2.4	Host Text Message Encoding (Host-To-Reader)	117
361	11.2.4.1	Vendor Extensions to Host Commands	117
362	11.2.5	Reader Text Message Encoding (Reader-To-Host)	117

363	11.2.5.1	Read Report Text encoding	117
364	11.2.5.2	Command Reply Message Format.....	117
365	11.2.5.3	Notification Text message encoding (Reader-To-Host)	117
366	11.2.6	Abbreviations.....	117
367	11.2.6.1	Object-Type Abbreviations.....	117
368	11.2.6.2	Command Abbreviations	117
369	12	Tag RF Protocol Bindings	117
370	12.1	EPCglobal Class0 RF Protocol Binding.....	117
371	12.1.1	TAG ID.....	117
372	12.1.2	Session	117
373	12.1.3	Data Field	117
374	12.1.4	Passwords	117
375	12.2	EPCglobal Class1 Gen1 RF Protocol Binding.....	117
376	12.2.1	TAG ID.....	117
377	12.2.2	Session	117
378	12.2.3	Data Field	117
379	12.2.4	Passwords	117
380	12.3	EPCglobal UHF Gen2 RF Protocol Binding.....	117
381	12.3.1	TAG ID.....	117
382	12.3.2	Session	117
383	12.3.3	Data Field	117
384	12.3.4	Passwords	117
385	A	XML Schemas	117
386	A.1	Common Types	117
387	A.2	Command Schema.....	117
388	A.3	Reply Schema	117
389	A.4	Notification Schema	117
390	B	References.....	117
391			

List of Figures

393	Figure 1: Layers of the Reader Protocol	18
394	Figure 2: Conceptual Reading Pipeline (non-normative)	21
395	Figure 3: Data Acquisition Timeout Across Read Cycles	23
396	Figure 4: Event Life Cycle.....	26
397	Figure 5: Reader Object Model.....	30
398	Figure 6: Additional Enumeration Objects	32
399	Figure 7: Concentrator Reader Architecture.....	35
400	Figure 8: Multiplexer Reader Architecture	36
401	Figure 9: Sources and Read Points	36
402	Figure 10: States of a serial transport connection.....	117
403	Figure 11: TCP Machine-to-Machine mode	117
404	Figure 12: XML Message Format, Read Report Schema.....	117
405	Figure 13: XML Message Format, Command Schema	117
406	Figure 14: XML Message Format, Reply Schema	117
407	Figure 15: XML Message Format, Notification Schema.....	117
408	Figure 16: XML Message Format, ACK/NAK Schema.....	117

List of Tables

409		
410	Table 1: Parameters to govern read trigger behaviour.....	23
411	Table 2: Terms relevant to Sources	34
412	Table 3: Trigger conditions.....	38
413	Table 4: Types of Events	40
414	Table 5: List of field names for Data Selector	43
415	Table 6: Default properties for Source object.....	74
416	Table 7: Possible Trigger values.....	101
417	Table 8: Default properties for Trigger object.....	102
418	Table 9: Default properties for TagSelector object.....	105
419	Table 10: Default properties for NotificationChannel object	110
420	Table 11: Default properties for DataSelector object	117
421	Table 12: Predefined tag field names.....	117
422	Table 13: Default properties for TagField object	117
423	Table 14: Basic Error Codes	117
424	Table 15: Sender Handshake fields.....	117
425	Table 16: Receiver Handshake fields	117
426	Table 17: Irrelevant commands for XML message format.....	117
427	Table 18: Mapping of data types in XML message format	117
428	Table 19: Mapping of enumeration objects for XML message format	117
429	Table 20: Allowed vendor extensions of XML message format	117
430	Table 21: Mapping of data types in Text message format.....	117
431	Table 22: Mapping of enumeration objects for Text message format.....	117
432	Table 23: Object type abbreviations for Text message format	117
433	Table 24: Command abbreviations of ReaderDevice object for Text message format .	117
434	Table 25: Command abbreviations of Source object for Text message format.....	117
435	Table 26: Command abbreviations of ReadPoint object for Text message format	117
436	Table 27: Command abbreviations of Trigger object for Text message format.....	117
437	Table 28: Command abbreviations of TagSelector object for Text message format	117
438	Table 29: Command abbreviations of NotificationChannel object for Text message	
439	format.....	117
440	Table 30: Command abbreviations of DataSelector object for Text message format ...	117

441	Table 31: Command abbreviations of EventType object for Text message format	117
442	Table 32: Command abbreviations of TriggerType object for Text message format....	117
443	Table 33: Command abbreviations of FieldName object for Text message format	117
444	Table 34: Command abbreviations of TagField object for Text message format	117
445	Table 35: RF-Protocol specific feature bindings	117

1 Introduction

This document defines Version 1.1 of the protocol by which tag readers interact with EPCglobal compliant software applications. The terms “tag reader” or “reader” include RFID tag readers, supporting any combination of RF protocols, fixed and hand-held, etc. It also includes readers of other kinds of tags, such as bar codes. Tag readers, despite the name, MAY also have the ability to write data into tags.

The Reader Protocol specifies the interaction between a device capable of reading (and possibly writing) tags, and application software. These two parties are herein referred to as the Reader and the Host. An example of a Host is an EPC-aware middleware or application, though the Reader Protocol itself does not require that any particular middleware or an application be used.

Note that the interaction between the Reader and RF tags is outside the scope of this specification. A goal of the Reader Protocol is to insulate the Host from knowing the details of how the Reader and tags interact. Readers MAY employ a variety of protocols to interact with tags (not exclusively RF tags: e.g., a Reader MAY be capable of reading optical bar codes), but the same Reader Protocol as specified herein is used between the Reader and Host.

The commands have been defined as SHALLs and CANs. The Reader Protocol Conformance Requirements will be developed to categorize readers by the functions they perform. Readers profiles will be developed to help conduct fair and accurate conformance tests. It is possible that existing CANs may change to reflect the profiles.

2 Terminology and Typographical Conventions

Within this specification, the terms SHALL, SHALL NOT, SHOULD, SHOULD NOT, MAY, NEED NOT, CAN, and CANNOT are to be interpreted as specified in Annex G of the ISO/IEC Directives, Part 2, 2001, 4th edition [ISODir2]. When used in this way, these terms will always be shown in ALL CAPS; when these words appear in ordinary typeface they are intended to have their ordinary English meaning.

All sections of this document are normative, with the exception of chapter 1, and where it is explicitly noted as non-normative.

The following typographical conventions are used throughout the document:

- ALL CAPS type is used for the special terms from [ISODir2] enumerated above.
- Monospace type is used to denote programming language, UML, XML identifiers and other coding constructs, as well as for the text of XML documents.
- Placeholders for changes that need to be made to this document prior to its reaching the final stage of approved EPCglobal specification are prefixed by a rightward-facing arrowhead, as this paragraph is.

3 Protocol Layers

The Reader Protocol is specified in three distinct layers, as illustrated below.

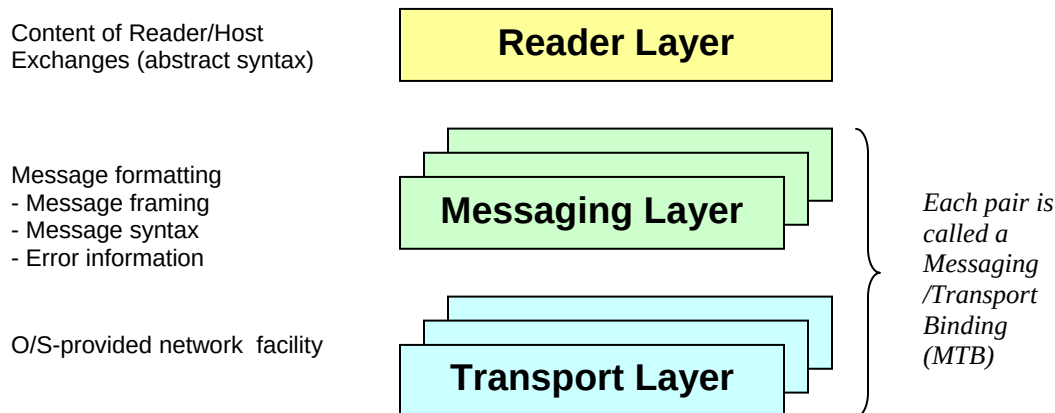


Figure 1: Layers of the Reader Protocol

The layers are:

Reader Layer This layer specifies the content and abstract syntax of messages exchanged between the Reader and Host. This layer is the heart of the Reader Protocol, defining the operations that Readers perform and what they mean.

Messaging layer This layer specifies how messages defined in the Reader Layer are formatted, framed, transformed, and carried on a specific network transport.

Transport Layer This layer corresponds to the networking facilities provided by the operating system or equivalent.

The Reader Protocol specification provides for multiple alternative implementations of the Messaging and Transport Layers. Each such permutation is called a Messaging/Transport Binding (MTB). Different MTBs provide for different kinds of transport, e.g., TCP/IP versus Bluetooth versus serial line. Different MTBs MAY also provide different means for establishing connections (e.g., whether the Reader contacts the Host or the Host contacts the Reader), initialization messages required to establish synchronization, and means for provisioning of configuration information.

Multiple standard MTBs are defined in this specification. Others MAY be defined and specified in separate, companion specifications.

3.1 Message Channels

The interface between a Host and a Reader is defined in terms of message channels, each representing an independent communication between Reader and Host. In particular, there are two types of message channels defined:

Control Channel The control channel carries requests issued by the Host to the Reader, and responses to these requests carried from the Reader to the Host. All messages exchanged on the control channel follow this request/response pattern.

Notification Channel The notification channel carries messages issued asynchronously by the Reader to the Host. Messages on the notification channel are only initiated by the Reader. The notification channel is primarily used to support a mode of operation in which the Reader asynchronously delivers tag reads to the Host, without the Host needing to poll.

Two channels are defined so that notifications MAY be directed to a different Host than the one currently issuing commands. This is facilitated through Control Channel commands that permit the Control Channel Host to specify a second Host to which the Reader connects to deliver notifications. In other cases, the Reader MAY permit notifications to be delivered to the same Host that issues Control Channel commands, with the traffic multiplexed on the same Transport Layer connection. Readers connected to Hosts via serial lines, for example, often require this mode of notification delivery.

The Reader Protocol supports a single or multiple Control Channels to a single Reader, and optionally one or more Notification Channels from a Reader to one or (potentially) more Hosts. The specific MTBs define exactly out how these permutations are handled logically as well as map to the underlying Transport Layer.

4 Reader Layer – Conceptual Overview

The Reader Protocol provides a uniform way for Hosts to access and control the conforming Readers manufactured by a variety of vendors. Different makes and models of readers MAY vary widely in the functionality they provide, from “dumb” readers that do little more than report what tags are currently within the reader’s RF field, to “smart” readers that provide sophisticated filtering, smoothing, reporting, and other functionality. The Reader Protocol defines a particular collection of features that are commonly implemented, and provides a standardized way of accessing and controlling those features when they are present. The Reader Protocol does not require that all readers implement all such features, only that when such features are provided, that they be made available to hosts in the manner specified herein.

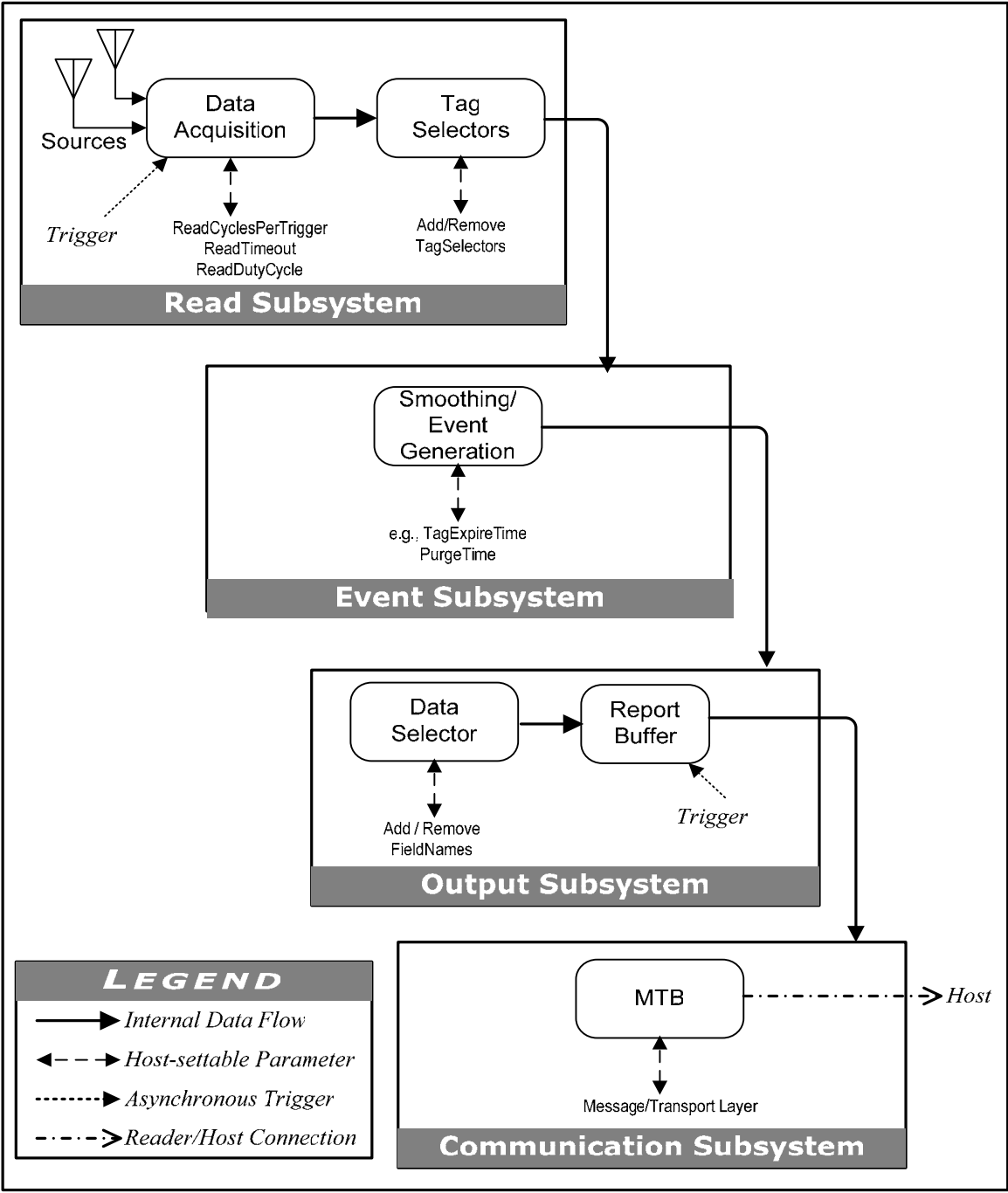
This section describes the features of a Reader in terms of a conceptual processing pipeline (see Figure 2 below). This perspective, the process pipeline, is intuitive and it therefore serves as a good introduction to a Reader. However, the pipeline model fails to adequately cover all possible complexities of a Reader. For this reason an object model is presented in the next section. The object model is helpful for understanding the many diverse and complex device configurations that are possible with advanced Readers.

The pipeline diagram below models the autonomous tag-reading functions of a reader as organized into several distinct processing stages. (Other reader functions, such as tag writing and “raw” tag reading, fall outside the scope of this particular diagram.) Tag read information flows from left to right in this diagram. Information about tag reads is made available to Hosts at certain stages. In some cases, this information is made available to the Host as a response to a command on the Command Channel (a “synchronous”

550 delivery of information), in other cases; the Reader sends the information autonomously
551 to the Host using the Notification Channel (an “asynchronous” delivery). Each stage also
552 has parameters that govern its operation; these parameters can be queried and set by the
553 Host via the Command Channel. For direct tag reading and writing operations, this
554 pipeline model doesn’t apply, however some functionality (e.g., TagSelectors) is reused
555 in the execution of these operations.

556 Conforming Readers are not required to provide every function that is implied by this
557 diagram. Moreover, some Readers MAY place more restrictions than others as to what
558 parameters can be set at each stage. This is another way the Reader Protocol accounts for
559 differences in functionality between particular Readers. For example, a Reader that
560 allows for an unlimited number of tag selectors to be set provides more functionality than
561 a Reader that permits only one tag selector, which in turn provides more functionality
562 than a Reader that permits no tag selectors at all. The Reader Protocol provides
563 commands (which all conforming readers SHALL implement) through which Hosts can
564 discover the capabilities of a particular reader.

565 It is important to note that the stages in the diagram are *conceptual and non-normative*,
566 and do not constrain the design of a conforming Reader. For example, some Reader
567 implementations MAY combine tag selectoring with data acquisition. In particular,
568 Readers that implement EPCglobal RF tag protocols can use tag selectors configured by
569 the host to reduce the time to execute the singulation part of the RF protocol, when the
570 specific filter patterns permit this to be done. While the design of such a reader does not
571 necessarily include a recognizable “data acquisition stage” distinct from a “tag
572 selectoring stage,” from the Host’s point of view (through the Reader Protocol) the
573 functional capability of the Reader is equivalent to a Reader that does have two distinct
574 stages. Similarly, a Reader does not have to be designed with distinct stages “Data
575 Selector”, “Report Buffer”, and “MTB”. A Reader MAY buffer all collected data
576 internally and wait to filter the data (i.e., apply data selection) until the data is being
577 formatted and communicated over a transport.



579
580

581 **Figure 2: Conceptual Reading Pipeline (non-normative)**

582

583 The following sub-sections provide a description of each stage of this conceptual pipeline
584 perspective.

4.1 Read Subsystem

This stage performs data acquisition and can be viewed as the input portion of the pipeline. It consists of **Sources**, **ReadTriggers** and **TagSelectors**. The Data Acquisition Stage is responsible for acquiring tag data from sources at specific points in time. The Reader Protocol provides parameters whereby Hosts MAY specify the frequency of data acquisition, how many attempts are made, triggering conditions, and so on. Each atomic interval in which the Data Acquisition Stage acquires data from one or more tags from a single source is called a *read cycle*. The Read Filtering Stage maintains a list of filter patterns configured by the host (defined in terms of **TagSelector** objects), and uses those patterns to filter out tags of no interest to the Host. The purpose of the filtering stage is to reduce the volume of data by only including tags of interest to the application.

4.1.1 Sources

Sources read tags and present their data to the Data Acquisition stage of the Read Subsystem in a Reader. Two examples of **Sources** are an RF antenna and a barcode scanner. In the next section describing the Reader object model (see section 5.2), the concept of **Sources** is further refined in terms of logical objects and physical devices.

4.1.2 ReadTrigger

The Reader Protocol provides for the initiation of “triggered” read operations by means other than an explicit host request. A **ReadTrigger** causes the Data Acquisition Stage to acquire data from sources. Triggering this activity causes data to flow from sources all the way to the report buffer, but does not cause any communication with the Host. See section 5.3 for details about the different conditions that can be used to establish a **ReadTrigger**.

4.1.3 Data Acquisition Stage

The Data Acquisition Stage corresponds to the logic in a conforming Reader that is responsible for sampling data from sources. Every source is presumed to have a smallest atomic unit of data acquisition, herein called a *read cycle*. For an EPCglobal RF Tag protocol, for example, a read cycle corresponds to one protocol run which attempts to read the tag IDs of all tags within the reader’s field. For a handheld bar code scanner, a read cycle might correspond to one attempt to read the bar code illuminated by the laser. The documentation for a conforming reader MAY specify exactly what is meant by a read cycle for each source it supports. In the case where a source corresponds to the direct reading of EPCglobal-compliant Class 1 RF Tags via an antenna, a read cycle SHALL correspond to a single protocol run.

Many readers are capable of performing several read cycles in succession in response to a read trigger. The Reader Protocol provides several parameters to model this capability. The parameters can be explained with reference to a time line:

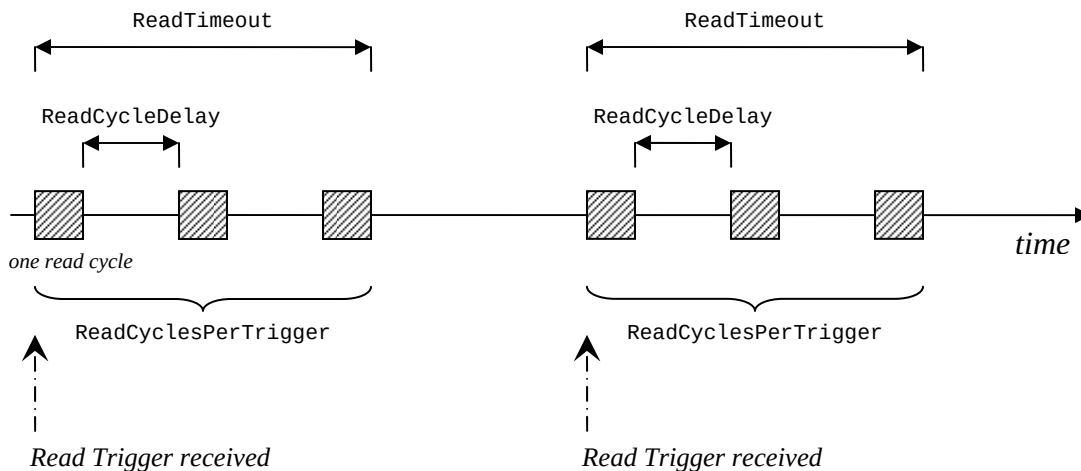


Figure 3: Data Acquisition Timeout Across Read Cycles

Each time a **ReadTrigger** is received, a series of read cycles is initiated. The following Host-settable parameters, govern the behavior when a **ReadTrigger** is received:

Parameter	Units	Meaning
ReadCyclesPerTrigger	count	The number of read cycles that SHALL be attempted, if the ReadTimeout does not expire first.
ReadTimeout	msec	The maximum amount of time in which to complete the specified number of read cycles. If zero, the read cycles MAY take an unlimited amount of time.
ReadMaxDutyCycle	Percentage	Maximum RF Active time as a percentage of overall time, as per the formula $\text{ReadMaxDutyCycle} = \text{ReadCycleDuration} / (\text{ReadCycleDuration} + \text{ReadCycleDelay})$ Compliant system MAY provide a means to control the maximum RF duty cycle. If implemented then the system MAY delay response to ReadTriggers in order to support duty cycle limitations to meet regulatory requirements.

Table 1: Parameters to govern read trigger behaviour

If a second or subsequent `ReadTrigger` is received before the complete series of read cycles for the first trigger has finished, then the series for the first trigger SHALL complete normally and then be followed by a new series of read cycles. This is intended to ensure that a full series of read cycles, with a complete protocol run per read cycle, SHALL follow each `ReadTrigger`.

4.1.3.1 Tag RF Protocol Binding

The Reader Protocol does not specify or even clarify the operation of tag RF protocols. To whatever extent possible, the Reader Protocol is designed to be independent of RF protocols. Nevertheless, to help insure Reader Protocol implementation interoperability, some aspects of the Reader Protocol are defined in terms of specific tag RF protocol features. These dependencies are detailed in section 12 which helps to minimize ambiguities regarding the interpretation of the Reader Protocol specification when applied to specific RF protocols.

4.1.4 Read Filter Stage

The Read Filter Stage corresponds to logic in a conforming Reader that removes certain tag reads according to their IDs. The Reader Protocol provides a way to describe a simple filtering scheme based on bitwise patterns. See section 5.4 for a detailed description how read filtering is defined in terms of `TagSelector` objects.

Readers that implement EPCglobal RF tag protocols MAY use `TagSelectors` configured by the Host to reduce the time to execute the singulation part of the RF protocol, when the specific filter patterns permit this to be done. This is an example of how the conceptual model of processing stages does not necessarily imply the order in which a conforming reader carries out its computations.

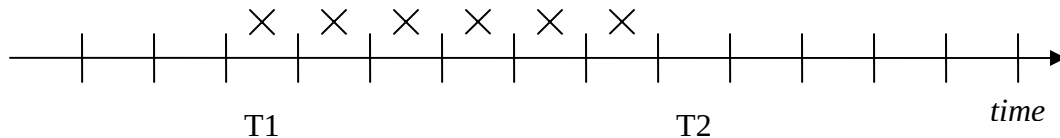
4.2 Event Subsystem

This stage performs event smoothing and generation. This stage MAY reduce the volume of data over time. When a given tag is present in the field of a particular source, the Read Subsystem includes that tag in its output each time a read cycle completes. When a tag appears present to a particular source for many read cycles, this generates a significant amount of data. The Event Generation Stage reduces this data volume by outputting an “event” only when something of interest happens: e.g., when the tag first appears and later when the tag is no longer present.

4.2.1 Smoothing/Event Generation Stage

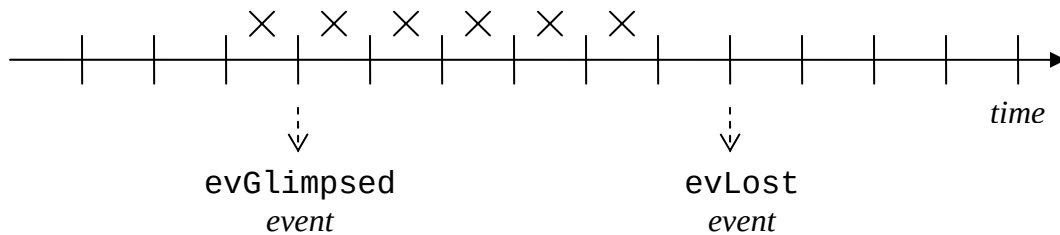
The Read Subsystem produces data every time a read cycle completes. Even with read filtering, this is often far more data than an application wishes to process. The purpose of the Event Subsystem is to model features available on many conforming Readers that serve to reduce this volume of data, by notifying the Host only about tags that have entered or left the view of a source.

Consider one particular tag (call it “X”) as seen by one particular source, which enters the source’s field at time T1 and leaves the source’s field at time T2. This can be visualized as follows:



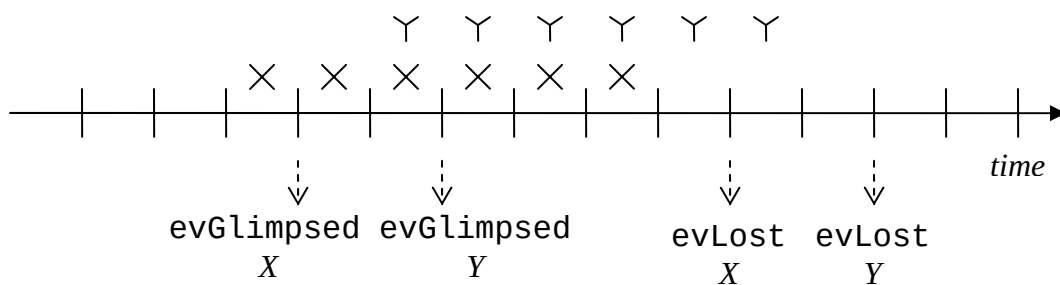
In this diagram, each gap between vertical lines represents a read cycle, and an X denotes a read cycle in which tag X is sensed. If a Host were receiving raw read data from the Read Subsystem, it would receive six reports that included X.

The Smoothing and Event Generation Stage is responsible for reducing this volume of data by generating “events” when the status of each tag changes. For example, the timeline above might generate two events, one when the tag is seen for the first time (designated by, *evGlimped.*) and another when the tag leaves the field and is no longer observed (designated by, *evLost*):



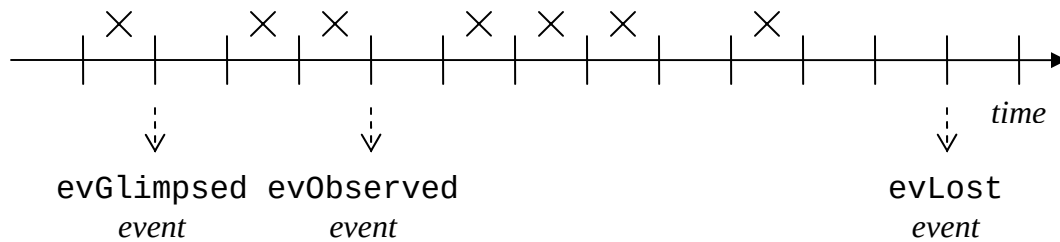
In this way, the volume of data has been reduced from six reports to two.

When generating events, individual tags SHALL be tracked separately, and the generated events SHALL indicate to what tag they pertain. Here is an example with two tags, X and Y:



It is clear from the above example that to do event generation, the Reader SHALL maintain some state information on a per-tag basis: in this case, the state information is whether the tag was seen in the previous read cycle or not. When multiple sources are involved, this information SHALL also be maintained per-source, so that the state information and events are maintained for each unique combination of tag and source. The state information is collectively referred to as the *tag list*.

RF tags have the characteristic that they are not always sensed by a Reader, even if they are within range of the Reader’s antenna. Thus, the timeline for a typical RF tag is probably not as clean as the diagrams above, but can look more as follows:



695

696 Here, there are three “gaps” in which tag X was not sensed, even though it was within the
 697 Reader’s field. If **evGlimpsed** and **evLost** events were generated as in the previous
 698 diagrams, there would be four **evGlimpsed** and four **evLost** events generated.

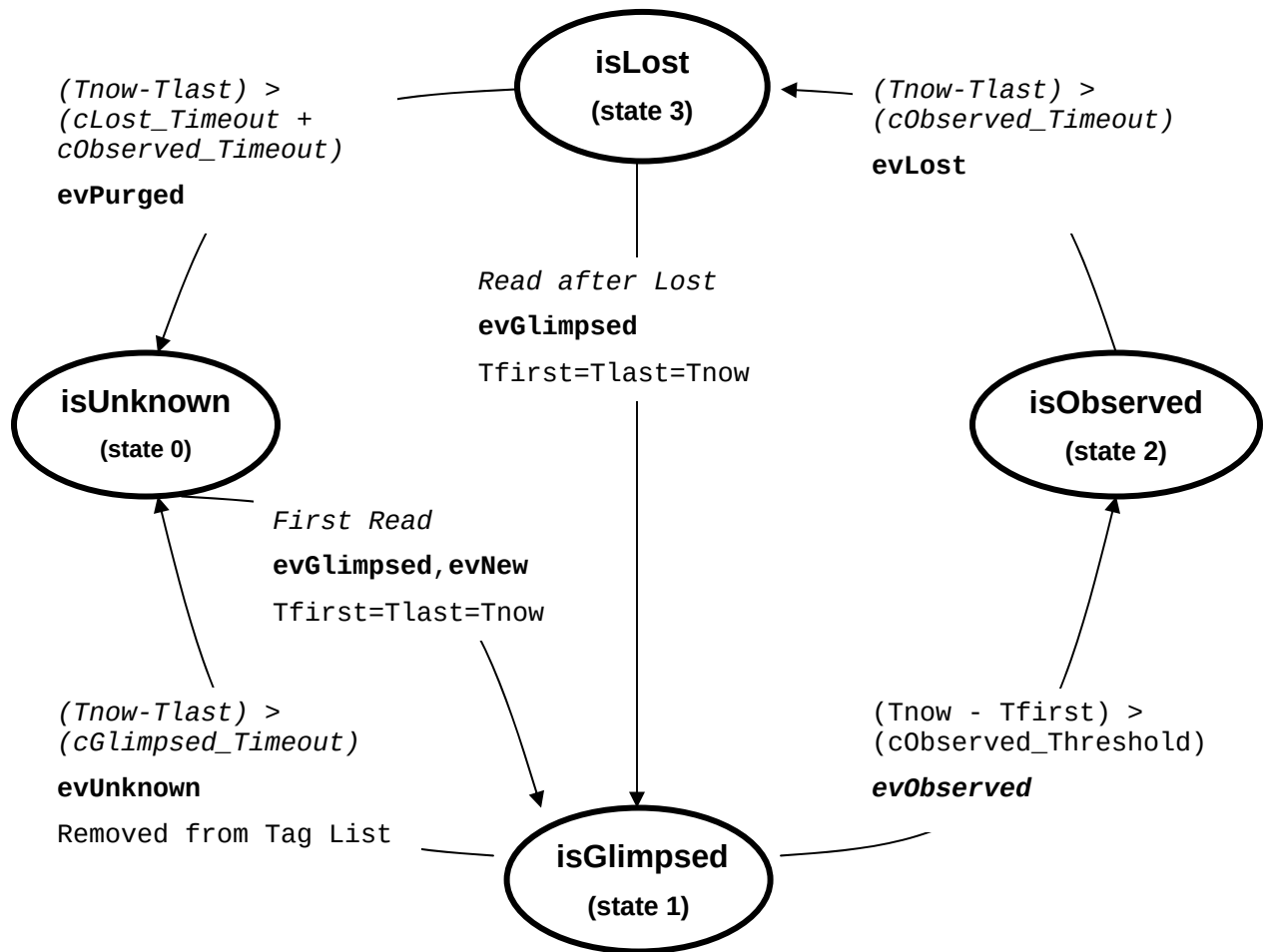
699 The “smoothing” aspect of the Smoothing and Event Generation Stage is designed to
 700 overcome these unwanted events. The idea is to generate a single **evGlimpsed** and
 701 **evLost** event as shown in the above diagram. Notice that the **evLost** event only
 702 occurs after the tag has been not present for several consecutive read cycles; this avoids
 703 generating unwanted events for the short gaps.

704 A new event, **evObserved**, is introduced to allow an application to discriminate against
 705 tags that enter the field only briefly, as compared to tags that remain for some period of
 706 time (some number of read cycles). Tags that are present only briefly could be spurious,
 707 such as a tag inadvertently sensed from an area beyond the area intended to be monitored
 708 by the Reader. It might be desirable that a tag that is present only briefly generate a
 709 **evGlimpsed** event, but *not* generate a subsequent **evObserved** or **evLost** event.
 710 A typical application might act only upon **evObserved** and **evLost** events.

711 These three events are part of a set that represents transitions between Reader
 712 “Knowledge-States” regarding the tag on a particular source. A useful model for the
 713 Reader’s knowledge about the tag includes four states (illustrated below): **isUnknown**,
 714 **isGlimpsed**, **isObserved** and **isLost**.

715 Movement between states is what defines an event. These transitions are controlled by a
 716 set of Host-configurable constants that are compared to the difference between the
 717 current time, (**Tnow**) and the time the reader first saw the tag (**Tfirst**) or most recently
 718 saw the tag (**Tlast**).

719 The States, Events and Constants used in the model are summarized in the following
 720 diagram:



721 **Figure 4: Event Life Cycle**

722

723 Every transition on this diagram MAY generate one or more events.

724 **Event Definitions:**

725 **evGlimpsed:** The transition between states 0 and 1 (**isUnknown** and
 726 **isGlimpsed**) or the transition between states 3 and 1 (**isLost** and
 727 **isGlimpsed**). A single read of the tag causes this to occur.

728 **evNew:** The transition between states 0 and 1 (**isUnknown** and **isGlimpsed**).
 729 This is the first time the Reader becomes aware of the tag. A single read of the tag
 730 causes this to occur.

731 **evUnknown:** The transition between states 1 and 0 (**isGlimpsed** and
 732 **isUnknown**). The tag is returning to the **isUnknown** state because it has not
 733 been seen for at least **cGlimpsed_Timeout** milliseconds.

evObserved: The transition between states 1 and 2 (**isGlimpsed** and **isObserved**). The tag has been seen for at least **cObserved_Threshold** milliseconds, with no intervening gap longer than **cGlimpsed_Timeout** milliseconds.

evLost: The transition between states 2 and 3 (**isObserved** and **isLost**). The tag had previously generated an **evObserved** event, but subsequently has not been seen for a time interval greater than **cObserved_Timeout** milliseconds.

evPurged: The transition between states 3 and 0 (**isLost** and **isUnknown**). The tag had previously generated a **evLost** event and the time since it was last seen recently seen (**Tnow** - **Tlast**) exceeds **cLost_Timeout** + **cObserved_Timeout**.

The values **cGlimpsed_Timeout**, **cObserved_Threshold**, **cObserved_Timeout**, and **cLost_Timeout** are parameters settable by the Host.

Note that if **cObserved_Threshold** is equal to zero, then a tag that is seen for the first time goes from **isUnknown** to **isGlimpsed** and immediately to **isObserved**. In this case, simultaneous **evNew**, **evGlimpsed** and a **evObserved** events SHALL be generated. Similarly, a tag in the **isLost** state that is seen again would move to **isGlimpsed** and immediately to **isObserved**. In this case, simultaneous **evGlimpsed** and a **evObserved** events SHALL be generated.

If **cLost_Timeout** is zero, then a tag that moves to the **isLost** state goes directly to **isUnknown**. In this case, both a **evLost** and a **evPurged** event SHALL be generated at the same time.

A conforming Reader NEED NOT provide every feature implied by this state diagram. For example, some readers can't distinguish between **isLost** and **isUnknown** tags. This will be modeled by restricting **cLost_Timeout** to always be equal to zero, and restricting the available events to include **evLost** but not **evPurged**. Likewise, a reader that does not distinguish between **isGlimpsed** and **isObserved** states SHALL restrict **cObserved_Threshold** to be equal to zero, and restrict the available events to include **evObserved** but not **evGlimpsed**. In general, Readers that provide less powerful smoothing functionality can be modeled by suitable restrictions on the smoothing parameters and/or limiting the events that can be reported.

4.3 Output Subsystem

This stage determines what data shall be communicated and manages the queuing (i.e., report buffering) of data in preparation for delivering it to the host.

Once output data is queued in the report buffer, there are two possible mechanisms for delivering it to the host: The Host can synchronously request delivery of all events in the report buffer, or alternatively, the events can be delivered asynchronously in response to a notification trigger.

773 **4.3.1 Data Selector Stage**

774 The Output Subsystem provides the ability for the host to control the specific contents of
775 the output that is stored in the report buffer and will be delivered by to the Host. This is
776 accomplished through the use of a Data Selector stage. A data selector contains a set of
777 “output fields” that a host is interested in, and associated protocol support that allows for
778 the discovery and selection of supported output fields.

779 Data selection is handled on two levels:

- 780 1) filtering events generated by the Smoothing and Event Generation Stage, and
- 781 2) establishing what data fields are reported.

782 A Data Selector only defines what data a host is interested in, but not how this selection
783 is implemented. A Reader Manufacturer MAY for example filter out events that are not
784 of interest before they are stored in the report buffer in order to save memory space.

785 A complete list of applicable output fields for data selectors can be found in section 0.

786 **4.3.2 Report Buffer Stage**

787 Events generated by the Smoothing and Event Generation Stage, and possibly filtered by
788 a Data Selector, are stored in a Report Buffer. The Host can synchronously request
789 delivery of all events in the report buffer, or the events can be delivered asynchronously
790 in response to various triggers.

791 When data has been delivered to the Host, the report buffer SHALL be cleared.

792 **4.3.3 Notification Trigger**

793 The Reader Protocol provides for the initiation of “triggered” asynchronous notifications
794 sent to the Host operations by means other than an explicit Host request. A Notification
795 Trigger causes the Report Buffer Stage to deliver queued reports to the host.

796 Section 5.3 provides details about the different conditions that can be used to establish a
797 Notification Trigger.

798 **4.4 Communication Subsystem**

799 After the Output Subsystem creates the desired report buffer contents, the data is
800 delivered to the Communication Subsystem for delivery to the host.

801 This stage performs data communication over the specified channels, using a specified
802 Message/Transport Binding (MTB). This stage formats internal data fields into data
803 representations according to the specifications of a particular binding.

804 **4.4.1 Message/Transport Binding Stage**

805 The Communication Subsystem consists of the specific Message/Transport Binding
806 (MTB) in use between the reader and the host. Different MTBs provide for different
807 kinds of transport, e.g., TCP/IP versus Bluetooth versus serial line. Different MTBs
808 MAY also provide different means for establishing connections (e.g., whether the Reader

809 contacts the Host or the Host contacts the Reader), initialization messages required to
810 establish synchronization, and means for provisioning of configuration information.
811 Multiple standard MTBs are defined in this specification. Others MAY be defined and
812 specified in separate, companion specifications.

5 Object Model

The simple sequence of the pipeline model described in the previous section is not sufficient to fully explain a Reader's complexity. This is because a reader MAY have multiple sources, multiple triggers, and multiple notification channels. Therefore, an object model is presented in this section that models the complex relationships between multiple sources, triggers, and channels.

The Reader Protocol shares the same object model as the specification defined by EPCglobal SAG working group "Reader Management", however, some objects and properties of the complete model are not relevant for Reader Protocol. The following figures show all objects of the complete model with only the properties relevant for Reader Protocol. Objects that are only relevant for Reader Management are shown in gray.

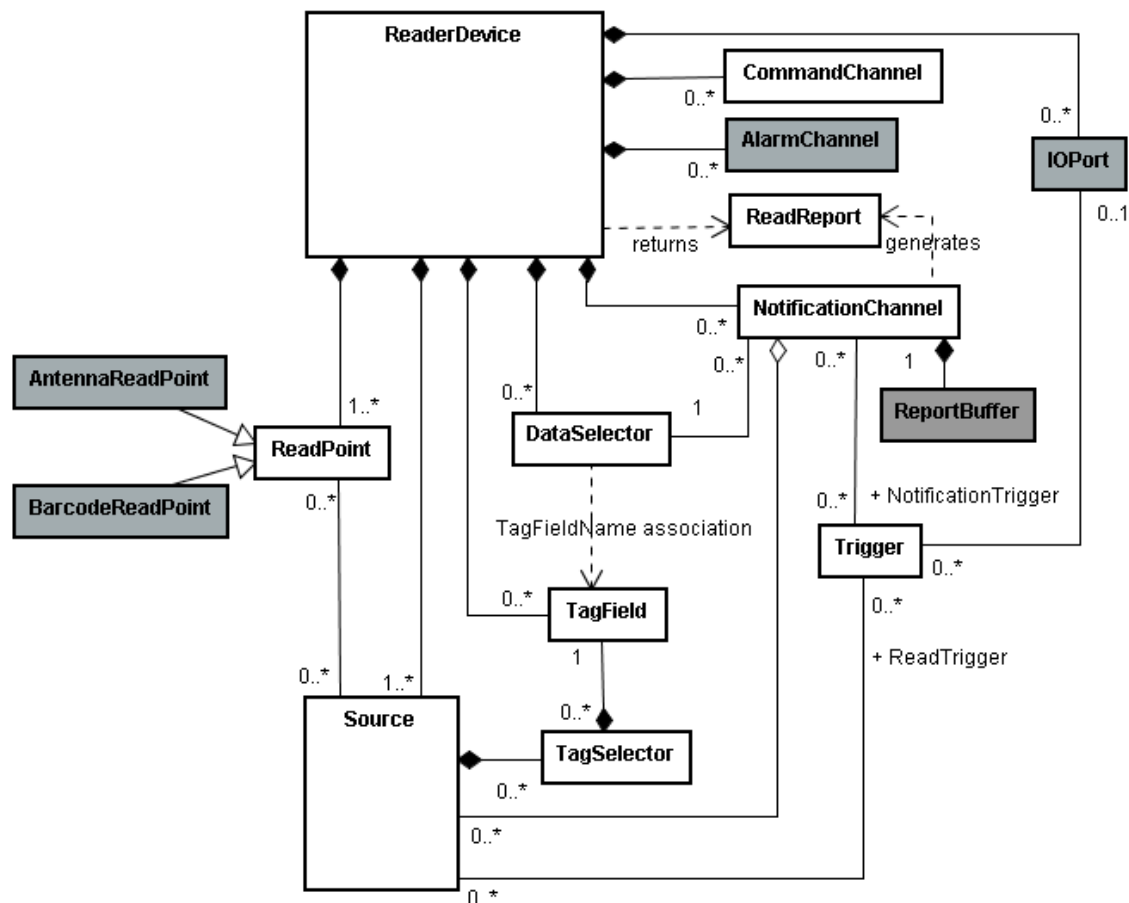


Figure 5: Reader Object Model

In this general model, a ReaderDevice has several Sources as well as multiple NotificationChannels. Each NotificationChannel is associated with exactly one DataSelector, which determines the parameters or data fields that are reported on this channel. Furthermore, a NotificationChannel only reports events for specific Sources. It therefore keeps a list of relevant Sources. Triggers can

832 be associated both with `Sources` and `NotificationChannels`. When associated
833 with a `Source`, a `Trigger` controls the execution of tag read cycles. When associated
834 with a `NotificationChannel`, a `Trigger` controls the asynchronous delivery of
835 queued reports.

836 Note that the `CommandChannel` object shown is a “virtual” object and only shown for
837 clarity. A command channel comes into existence whenever a Host opens the respective
838 communication channel and the Reader doesn’t refuse it. There are no explicit operations
839 on this object. A compliant Reader SHALL however have at least one command channel
840 with a default address where it is listening to incoming commands. This default address
841 SHALL be listed in the documentation of a compliant reader.

842 Similarly, the `ReportBuffer` object exists only implicitly. While it is necessary for
843 buffering and reporting reads and events, there are no explicit operations on it and it is up
844 to the reader vendor how to implement a report buffer. Note that the `ReportBuffer`
845 object is associated with the `NotificationChannel`, meaning that conceptually
846 there is one `ReportBuffer` per `NotificationChannel`. The reason for this is
847 simply that a `NotificationChannel` SHALL be able to queue data that it forwards
848 to the host. The `NotificationChannel`’s associated `ReportBuffer` is its
849 queuing mechanism and its queue SHALL be separate from other
850 `NotificationChannel` queues.

851 The structure of the `ReadReport` object is defined separately and is MTB-specific, see
852 chapter 1 for details.

853 In addition, there are several enumeration objects:

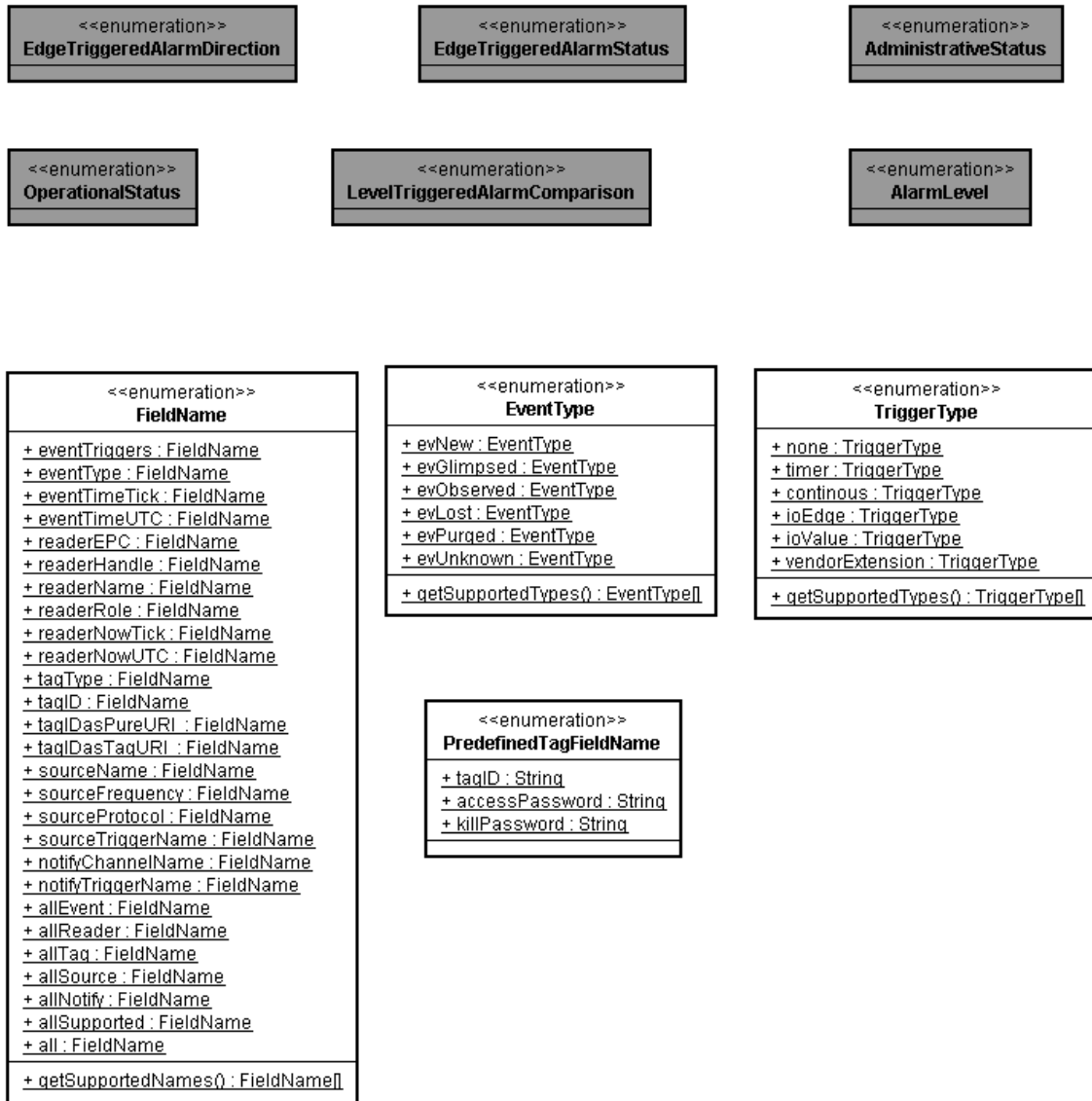


Figure 6: Additional Enumeration Objects

A compliant Reader NEED NOT implement an object model internally, as long as the command messages can be correctly interpreted and executed and notifications can be generated.

The following sections will define these objects in more detail.

5.1 ReaderDevice Object

The ReaderDevice object is the base container for most other objects in the Reader object model including at least one pre-configured CommandChannel object. The

866 ReaderDevice object also contains several attributes that are used to manage the
867 Reader device and, for network Readers, including attributes used to manage the
868 Reader's network interface.

869 **5.2 Sources**

870 Source objects are contained by the ReaderDevice object and by
871 NotificationChannel objects. This section describes both the rationale for
872 Source objects and their relationship to other objects in the Reader object model.

873 A Reader that supports multiple input sensors can be required to select and unselect (i.e.,
874 enabled or disabled) individual sensors. Moreover, it might be necessary for a multi-
875 input sensor Reader to manage two or more distinct applications. This is sometimes
876 required to minimize overall system cost. As an example of a reader managing two
877 distinct applications, input A is assigned to an antenna mounted over a dock door and
878 input B is assigned to an antenna mounted over a conveyor. In this scenario input A and
879 input B will require different ReadTriggers, TagSelectors, Smoothing
880 parameters, and possibly different EventFilters.

881 In the Reader object model, a Source object can encapsulate one or more input sensors
882 called ReadPoints. A ReadPoint can be any physical entity that is capable of
883 acquiring data. A single RF tag reader antenna is a simple example of a ReadPoint.
884 ReadPoints are not limited to antennas, however: for example, a bar code scanning
885 wand is another example of a ReadPoint.

886 A Source is a logical entity with zero, one, or more ReadPoints assigned to it. For
887 example, a reader might define a Source that represents tags read on either of two
888 antennas. In general, a Reader segregates tags read according to Source to provide
889 applications with some clue about the external situation in which the tag was sensed. A
890 Reader MAY come from a manufacturer with one or more *pre-configured* Sources. A
891 pre-configured Source MAY be *fixed*, meaning that users cannot add or remove its
892 ReadPoints. Different readers MAY vary widely on what Sources and
893 ReadPoints are available. The Reader Protocol provides commands for discovering
894 the number and names of available Sources, ReadPoints and their corresponding
895 names.

896 The attributes MaxReadDutyCycle, ReadCyclesPerTrigger and
897 ReadTimeout, as well as the term Read Cycle, apply at the Source-level. How these
898 are applied to the individual ReadPoints, and how one cycles through the different
899 ReadPoints in one Read Cycle is implementation dependent and not specified here,
900 since different situations may require different algorithms to achieve the best reading
901 performance.

902 A Source object then, is a container for ReadPoint objects. A Source object is
903 also a container of TagSelectors and ReadTriggers, and it maintains
904 configurable attributes that manage data smoothing and event generation. Each Source
905 object maintains a separate set of data smoothing parameter values. The output produced
906 by a Source is Events. Events generated by a Source are sent to all
907 NotificationChannels containing the Source. Note that state for the event
908 generation is kept per tag and per Source, but not per ReadPoint.

909 The following table lists terms relevant to Sources:

Term	Description
Allocated	A Reader SHALL be able to report the number of Source objects it supports. A Source object that has been named and added to the ReaderDevice object is said to be allocated.
Deallocated	Once a Source object is removed from the ReaderDevice object the Source object is said to be deallocated.
Name	When a Source object is allocated (i.e., added to the ReaderDevice object) it is given a unique name by which it can later be referenced.
Assigned	When a ReadPoint is added to a Source it is said to be assigned.
Unassigned	A ReadPoint that is not assigned to a Source is said to be unassigned.
Pre-configured	A Source that comes allocated by the Reader manufacturer is said to be pre-configured. A pre-configured Source will usually also have one or more ReadPoints assigned to it.
Fixed	A pre-configured Source that cannot have its ReadPoint assignment modified (i.e., ReadPoints cannot be added or removed). Obviously, to be useful a fixed Source SHALL come with at least one ReadPoint assigned to it.
Current Source	The Reader maintains a reference to a Source object that is used by default when processing a command that requires a Source but that is omitted from the command parameters provided by the Host.

910 **Table 2: Terms relevant to Sources**

911 **5.2.1 Source Organization**

912 A Reader MAY be configured to support multiple Sources.¹ Any Source MAY be
913 pre-configured by the Reader manufacturer. A pre-configured Source MAY be
914 implemented to be fixed (i.e., its ReadPoints cannot be re-configured). When only a
915 single Source is present, the operations requested through the Reader Protocol apply to
916 that single Source without ambiguity. In this case, the Reader's Current Source
917 SHALL always reference the single Source object.

918 When multiple Sources are available, the Reader SHALL provide the means to select
919 what Sources SHALL be used to acquire data at any given time and the Reader MAY
920 provide the means of applying different configuration parameters (e.g.,
921 ReadTriggers, TagSelectors, etc.) to the different Sources.

¹ Vendors are free to choose what (if any) Sources and other objects are pre-configured in 'factory fresh' configurations. There is no standard requirement for a Reader to be preconfigured with any Source definition(s); that is a product definition choice, not a specification requirement.

Explanation (non-normative): the Source and ReadPoint structure is intended to accommodate many possible reader architectures, which may be informally classified as follows:

- *Single ReadPoint*

This device will usually have just one Source pre-configured with the ReadPoint assigned to the Source. All operations requested through the Reader Protocol apply to that Source.

- *Concentrator*

See Figure 7: Concentrator Reader Architecture. This is a device that is connected to two or more Sources where each Source has embedded processing capability that enables the device to execute a read cycle on one Source concurrently with the execution of read cycles on other Sources.

- *Multiplexer*

See Figure 8: Multiplexer Reader Architecture. This is a device that is connected to two or more Sources where each Source has no processing capability (e.g., an antenna). This device architecture requires that the device's processor is multiplexed among the read cycles of the different Sources. In this configuration, the device's processor can provide separate configuration parameters for each Source.

- *Hybrid*

A device can be architected to include a hybrid of embedded Sources where some Sources have embedded processing capability and other Sources do not.

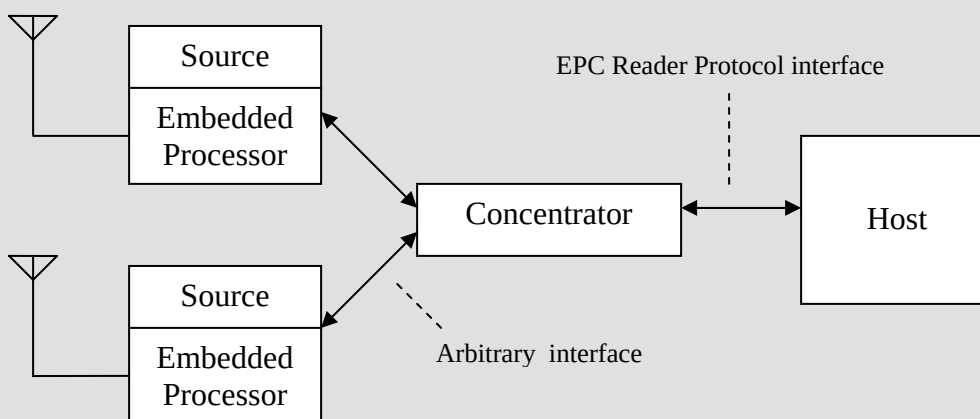


Figure 7: Concentrator Reader Architecture

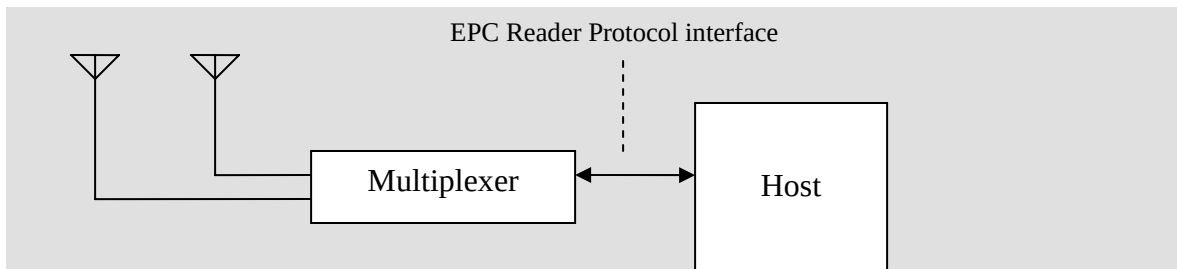
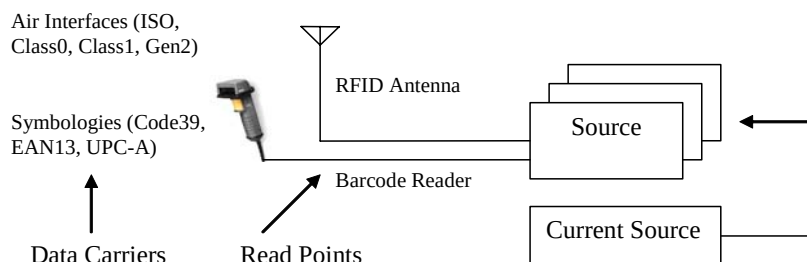


Figure 8: Multiplexer Reader Architecture

*An important distinction between the concentrator and multiplexer architectures is that the processors associated with **Sources** connected to a concentrator can execute read cycles concurrently with one another. The processor in the multiplexing control device is shared between the **Sources**.*

923

924 The Reader Protocol defines several features for Readers that support multiple sources.
925 These features are based upon the model illustrated below:



926

927 **Figure 9: Sources and Read Points**

928 The data carriers and **ReadPoints** that a Reader supports are dependent on the physical
929 device characteristics of the Reader. Data carriers are identified by the defining standards
930 organization and the data carrier standard (e.g., EPCglobal, UHF Gen2). A **ReadPoint**
931 can support one or more data carriers. A Reader's **ReadPoint** identifiers are pre-
932 configured by the Reader manufacturer. A Reader **SHALL** support at least one **Source**.
933 To minimize Reader requirements, a single **Source** can be supported that comes from
934 the manufacturer pre-configured and fixed (i.e., its **ReadPoint** assignments cannot be
935 dynamically re-configured by a host).

936 The number of **Sources** that a Reader supports is pre-configured by the Reader
937 manufacturer (e.g., a pool of **Sources**). A Reader manufacturer **MAY** pre-configure
938 **Sources** with a unique name, and it **MAY** allow a Host to specify additional
939 **Sources**. Each **Source** **SHALL** have a unique, but otherwise arbitrary identifier (i.e.,
940 name).

941 The Reader Protocol also defines the concept of a **Current Source** and it provides a
942 mechanism for associating the **Current Source** with a particular **Source**. Reader

commands that take a **Source** input parameter SHALL use the **Current Source** when the **Source** input parameter is omitted.

Data carrier identification (i.e. tag type) and **Source** are types of information that can be reported with the tag data.

Readers SHALL be configured in one of the topologies outlined here:

- **Zero ReadPoints**

This configuration has no functional utility. This case will most likely occur with the intention of later upgrading the device with one or more **ReadPoints**. Nevertheless, a Reader with zero **ReadPoints** MAY be accessed and queried for Reader information.

- **One ReadPoint**

- ❖ **One Source**

A Reader with just one **ReadPoint** SHALL have just one **Source** that is pre-configured to be allocated with a pre-defined name and with the one **ReadPoint** assigned to it.

- **Multiple ReadPoints**

- ❖ **One Source**

A Reader that supports multiple **ReadPoints** will likely support at least one **Source** that can be dynamically configured so that the Host can select which **ReadPoints** will be used in read cycles. A Reader with just one **Source** SHALL come pre-configured with this **Source** already allocated with a pre-defined name.

ReadPoints can be assigned to the **Source** by Host commands. In effect, a **ReadPoint** that is not assigned to a **Source** is disabled.

- ❖ **Multiple Sources**

A Reader that supports multiple **ReadPoints** and **Sources** can be used to implement multiple applications that are defined in terms of different **ReadTriggers**, **TagSelectors**, etc.

Readers that support multiple **ReadPoints** and multiple **Sources** MAY prohibit **ReadPoints** from being assigned to multiple **Sources** simultaneously (i.e., **ReadPoints** cannot be shared). If a Reader permits shared **ReadPoints**, the treatment of **ReadPoint** concurrent access is implementation dependent.

5.3 Triggers

Trigger objects can be contained by a **Source** object where they act as **ReadTriggers**. Or, **Trigger** objects can be contained by a **NotificationChannel** object where they act as **NotificationTriggers**. This section describes both the rationale for **Trigger** objects and their relationship to other objects in the Reader object model.

983 The Reader Protocol provides for the initiation of Read and Reporting operations by
984 means other than an explicit Host request. A `ReadTrigger` causes the Reader to
985 acquire data from the `Trigger` object's associated `Source`. Triggering a read causes
986 data to flow from the `Source` to a `ReportBuffer`, but does not cause any
987 communication with the Host.

988 A `NotificationTrigger` causes the Reader to send the entire contents of the
989 `ReportBuffer` which is associated with this `Trigger`'s `NotificationChannel`.

990 The Reader Protocol provides a variety of ways for the Host to specify trigger conditions.
 991 These include:

Trigger	Description	Host-specified Parameters
continuous	The activity is triggered as rapidly as possible. For ReadTriggers, regulatory limits on duty cycles SHALL be taken into account. For NotificationTriggers, this means a Notification SHALL be sent whenever a new Event appears in the Report Buffer.	[none]
timer	The activity is triggered at regular time intervals.	Time in msec between triggers.
ioEdge	The activity is triggered by a 0→1 transition or 1→0 transition of a binary-valued external signal. Signals are named by specifying a numbered “IO Port”, and a numbered “IO Pin” within that port. The documentation for a conforming reader SHALL specify to what connectors or other physical elements each port and pin setting correspond. Ports and pins need not literally correspond to physical I/O pins on connectors; they MAY instead correspond to any binary-valued, externally generated quantity. For example, a reader might respond to the arrival of a particular kind of network packet, and expose this as a possible trigger by assigning a port and pin number to it.	Which transition: 0→1 or 1→0. Which IO Port Which pin of the IO Port
ioValue	The activity is triggered by the detection of a specific value on a specified I/O port. As with IO edge triggers, the port need not literally correspond to a physical I/O port; it MAY instead correspond to any integer-valued, externally generated quantity.	Which IO Port What value constitutes a trigger and a mask indicating which bits to include in the trigger.
<VendorName>: <Any>	Vendors can specify proprietary triggering mechanisms. The name of these triggers SHALL have the vendor name and a colon as prefix.	Arbitrary.

992 **Table 3: Trigger conditions**

Conforming Readers MAY provide any or none of the triggers described above. The Reader Protocol includes commands whereby a Host MAY discover what triggering modes are available.

Besides the autonomous triggers defined above, the host MAY issue a command to force the instigation of a triggered activity. This is slightly different than the host issuing a synchronous command. For example, the synchronous `ReadIDS` command initiates a flow of information from a `Source` all the way to a `ReportBuffer`, and then the contents of the `ReportBuffer` are delivered to the Host as the response to the command. The command is “synchronous” because the command does not complete until the reading is complete and the reports are returned to the host. If the host issues a `Trigger.fire` command, in contrast, information flows from a `Source` to a `ReportBuffer`, but no report is sent to the host; moreover, the `Trigger.fire` command MAY return its reply to the Host even before reading is complete.

5.4 TagSelectors

A `TagSelector` object is contained by the `Source` object. This section describes both the rationale for `TagSelector` objects and their relationship to other objects in the Reader object model.

A `TagSelector` encapsulates the logic in a Reader that eliminates tags from being reported according to the conditions specified by the `TagSelector` parameters. This *filtering logic* is a simple scheme based on bit-wise patterns.

A `TagSelector` is specified by using two hexadecimal strings, a filter value and a filter mask. In the filter mask `M`, all bit positions where the value is important for the filtering SHALL be set to 1. In the filter value `V`, the desired value of the bit positions defined as relevant in `M` can be set.

A tag ID matches the filter if and only if the result of applying the filter mask on the filter value using a bit-wise AND operation is the same as when applying the filter mask on the tag ID:

If $((V \text{ bitand } M) == (A \text{ bitand } M))$ Then <code>TagIDMatchesTheFilter()</code>
--

Example:

filter mask <code>M = 1C</code> (00011100 in binary, meaning we are only interested in the bit values at positions 4, 5, and 6)
filter value <code>V = 10</code> (00010000 in binary. Because of the setting of <code>M</code> , only the values of bit positions 4-6 are important. Therefore in this case a filter value of “F3” / 11110011 would have the same result)
Actual tag ID data <code>A = 55</code> (01010101 in binary)
In this case: $V \text{ bitand } M = 10 \text{ bitand } 1C = 00010000 \text{ bitand } 00011100 = 00010000 = 10$ $A \text{ bitand } M = 55 \text{ bitand } 1C = 01010101 \text{ bitand } 00011100 = 00010100 = 14$ The two values are different, so there is no match.

1036

1037 In addition to this filter definition, a TagSelector contains a TagField object. The
1038 TagField defines the tag data field for which the TagSelector applies (i.e., the TagField
1039 specifies where to find the data field on the tag that should be processed by the filter).

1040 Readers MAY use TagSelectors configured by the Host to reduce the time to
1041 execute the singulation part of the tag's RF protocol, when the specific filter patterns
1042 permit this to be done. This is an example of how the conceptual model of processing
1043 stages does not necessarily imply the order in which a conforming reader carries out its
1044 computations.

1045 **5.4.1 Multiple Tag Selectors**

1046 Multiple TagSelector objects can be associated with any given source. Each of these filter
1047 objects is specified to be either *inclusive* (meaning that only tags matching the filter
1048 SHALL be reported) or *exclusive* (meaning that a tag SHALL only be reported if it does
1049 not match the filter).

1050 If multiple TagSelectors are used, a tag SHALL be reported only if the following two
1051 conditions hold:

- 1052 • The tag matches at least one of the inclusive patterns; *and*
- 1053 • The tag does not match any of the exclusive patterns.

1054 As a special case, if zero inclusive patterns are defined, the first check SHALL be
1055 omitted.

1056 **5.5 Events**

1057 Event objects are generated by Source objects. Events are sent to all
1058 NotificationChannel objects associated with the generating Source object.

1059 Events defined by the Reader Protocol are listed below and described in detail in section
1060 4.2.1.

Event	Description
evGlimpsed	A single read of the tag causes this to occur.
evNew	This is the first time the Reader becomes aware of the tag.
evUnknown	The tag is returning to the <code>isUnknown</code> state because it has not been seen for a specified period of time.
evObserved	The tag has been seen for a minimum time with no intervening gap longer than another specified time.
evLost	The tag had previously generated an <code>evObserved</code> event, but subsequently has not been seen for a specified time.
evPurged	The tag had previously generated an <code>evLost</code> event and the time since it was last seen recently seen exceeds a specified time.

<VendorName> :	Arbitrary vendor-specified event. The name of these events SHALL
<Any>	have the vendor name and a colon as prefix.

Table 4: Types of Events

A conforming Reader implementing notification channels SHALL generate at least one of either two event types: `evGlimpsed` or `evNew`. Readers MAY generate other events including vendor-specified events.

5.6 Channels

`Channel` objects are responsible for maintaining communications between the Reader and the Host. There are two types of `Channel` objects: `CommandChannels` and `NotificationChannels`. Both `CommandChannel` and `NotificationChannels` objects are contained by the `ReaderDevice` object. A `Channel` implements a particular message/transport binding (MTB). A binding includes both a particular message content format and transport interface. The Reader Protocol currently specifies both Text and XML message content formatting. However, subsequent versions of this specification are expected to also support binary formatting. Transports included in the current specification are: serial, TCP, and HTTP.

The Reader Protocol defines a handshake protocol (see section 10.1) that a Host and Reader can use to establish what binding SHALL be used for communications.

As the name implies, a `CommandChannel` object is responsible for receiving commands from a Host. A `CommandChannel` connection SHALL always be initiated by a Host. A Reader MAY be controlled by more than one host (e.g., a Host managing data collection and another Host managing hardware devices). A `CommandChannel` is always associated with a single Host. For this reason, a Reader MAY support multiple `CommandChannels` and thereby support multiple controlling Hosts. A compliant Reader SHALL contain at least one `CommandChannel`. All supported `CommandChannels` SHALL be pre-configured (e.g., the address to which a host can connect) and allocated to the `ReaderDevice` object by the Reader manufacturer. Note that while there can be several `CommandChannels`, there is only one Reader state. That is, if Host A issues a `setFoo` command over its command channel, then Host B will see the change if it issues a `getFoo` command over its own command channel.

A `NotificationChannel` object is responsible for communicating information to a host asynchronously. `NotificationChannels` SHALL be created by a Host via a `CommandChannel` request. This request (`NotificationChannel.create`) SHALL specify a connection address, which includes the specification (see section 9.1 for a description of the address URI-style address syntax) of the transport to be used. The Reader MAY initiate the `NotificationChannel` connection with a Host. Or, the Reader MAY listen for a `NotificationChannel` connection initiated by a Host.

5.7 DataSelectors

DataSelector objects are either contained by NotificationChannel objects or provided as a parameter in commands to read tags. It provides definitions for both event filtering and data field selection.

If a DataSelector has some fields for which there is no information or that are inapplicable (e.g., a trigger name when no triggers are used), these fields SHALL be ignored and not included in the output.

The following table defines the names of all known data fields, describes their meaning, and lists the compliance requirement.

Data field	Compliance	Description
eventTriggers	MAY	All read triggers that lead to the generation of this event
eventType	MAY	The type of the event, e.g. evGlimpsed.
eventTimeTick	MAY	The time the event occurred in ticks.
eventTimeUTC	MAY	The time the event occurred in UTC.
readerEPC	SHALL	The ReaderEPC attribute of the reader.
readerHandle	MAY	The Reader Handle attribute of the reader.
readerName	SHALL	The Reader Name attribute of the reader.
readerRole	SHALL	The Reader Role attribute of the reader.
readerNowTick	MAY	The time at the reader when the notification is sent, in ticks.
readerNowUTC	MAY	The time at the reader when the notification is sent, in UTC
tagType	MAY	The tag type, e.g. ISO, Gen2.
tagID	SHALL	The ID of the tag in raw binary format.
tagIDasPureURI	MAY	The ID of the tag represented as a <i>pure identifier</i> according to the EPC Tag Data Standards specification [TDS1.1]. Example: urn:epc:id:sgtin:0652642.800031.400 If the tag ID is not a valid EPC, then the raw URI form SHALL be used, e.g urn:epc:raw:64.20018283527919

Data field	Compliance	Description
tagIDasTagURI	MAY	The ID of the tag represented as a <i>tag URI</i> according to the EPC Tag Data Standards specification [TDS1.1]. Example: urn:epc:tag:sgtin-64:3.0652642.800031.400 If the tag ID is not a valid EPC, then the raw URI form SHALL be used, e.g urn:epc:raw:64.20018283527919
sourceName	MAY	The name of the source from which this tag read came.
sourceFrequency	MAY	The frequency the specific source uses.
sourceProtocol	MAY	The (air) protocol the specific source uses.
notifyChannelName	MAY	The name of the channel for this notification.
notifyTriggerName	MAY	The notify trigger that lead to this notification being sent.
allEvent	MAY	All event... fields.
allReader	MAY	All reader... fields.
allTag	MAY	All tag... fields.
allSource	MAY	All source... fields
allNotify	MAY	All notify... fields.
all	MAY	All fields.
allSupported	SHALL	All fields that are supported by this reader. Which set of fields this is, is vendor-dependent. This is also the set of fields that SHALL be reported when no DataSelector is defined.
<Vendorname>:<Any>	MAY	Vendors MAY define additional fields. The name of these fields SHALL have the vendor name and a colon as prefix.

Table 5: List of field names for Data Selector

In addition, TagFieldName objects can be added to a data selector to have specific memory fields on the tag read and reported. If several TagFieldName objects are added, all these fields SHALL be read.

5.8 TagFields and TagFieldValues

TagField objects can be defined to read or write specific memory fields on the tag, e.g. data in the user memory bank of a Gen2 tag. The TagField object is used to abstract from physical tag memory layout so that applications can use logical tag field names instead.

5.9 Object Names

Each object SHALL be identified by an arbitrary, but unique name, settable by the host. A name is unique per object-type, meaning that there SHALL NOT be any two objects of the same type with the same name, but there MAY be two objects of different types with the same name. The latter is not advisable, however.

For example, if a Source “myObject” exists, it SHALL NOT be possible to create another Source also called “myObject”. However, a TagSelector called “myObject” can be created, assuming that such a TagSelector doesn’t exist yet.

6 Reader Layer – Commands

This section specifies messages exchanged at the Reader Layer.

As described in section 3, messages on the control channel follow a request/response pattern, where the Host sends a request message, and the Reader responds later with a response message (be it a normal response or error response). Messages on the notification channel are sent asynchronously from Reader to Host. Some control messages MAY serve as triggers for notification messages.

Commands are described below in an abstract manner using a UML-like notation. The commands are grouped by the object they belong to. See chapter 5 for an overview of all objects. Note that while the description here follows an object-oriented model, vendors are not forced to use an object-oriented model in their implementations. Each MTB defines how the information about the object is transported, e.g. by adding an additional parameter with the name of the object.

Conforming readers SHALL support all mandatory commands and MAY support all optional commands. Vendor-specific commands may be enabled in conformance with this specifications but are not specified herein. A reader shall use vendor-specific commands only in accordance with the rules defined in the specification. A custom command SHALL NOT solely duplicate the functionality of any mandatory or optional command defined in this specification by a different method.

This section documents the normative definition of Reader Protocol commands and the compliance requirements for each command. Other sections document Message/Transport Bindings (MTB) for Reader Protocol commands. If there is an inconsistency between compliance requirements documented in this section and requirements documented in other sections of the Reader Protocol Specification, then this section (Reader Layer – Commands) is the overriding definition of requirements.

There are the following type of commands:

- DO-type commands that cause the Reader to take action.
- SET-type commands that cause the Reader to change internal state variables.
- GET-type commands that cause the Reader to return data to the Host on the command channel.

In short, ‘properties’ or ‘attributes’ (aka variables) with SET and/or GET operations, and ‘functions’ or ‘methods’ performing some action.

The maximum length of settable string attributes is vendor-dependent, but all readers SHALL support a length of at least 64 bytes. Attempts to set a string of excessive length SHALL result in an error.

All commands are *atomic*, meaning that they SHALL be executed either completely or not at all. If there is an error while trying to execute a command, the state of the reader including all attribute values of all objects SHALL remain unchanged. For example, when adding arrays of things (e.g., `FieldNames` to a `DataSelector`), and one value

1167 is not supported or not known, then no values SHALL be added and an error SHALL be
1168 raised.

1169 The return values of the command indicate the return values if the command was
1170 successfully executed. If the command cannot be executed correctly, the return value
1171 SHALL be undefined and an error condition SHALL be raised. The error conditions are
1172 explained in chapter 8.

1173

1174 **UML-like notation for an abstract command:**

```
1175 [static] OBJECT.COMMAND ([  
1176             PARAMETER: DataType,  
1177             PARAMETER: DataType,  
1178             ...]) : DataType
```

1179

1180 The `static` keyword indicates that a command doesn't require an object instance in
1181 order to be executed.

1182

1183 **DataTypes for command parameters and command return values:**

1184 • **address**

1185 The address, where e.g. a NotificationChannel is sending its reports to. Uses URI-
1186 style formatting, e.g.:

```
1187     tcp://foo.bar.com:2000  
1188     serial://com1:3  
1189     http://myreaderisawebserver:7183
```

1190 See section 9.1 *Address Notation* for more details.

1191 • **epc**

1192 An EPC. The formatting depends on the MTB; it MAY be a string in urn-notation or
1193 also the raw binary value.

1194 • **binary**

1195 A binary value of variable length. The formatting is MTB-specific.

1196 • **boolean**

1197 A boolean value of True or False. The formatting is MTB-specific.

1198 • **hexstring**

1199 A string of variable length containing a hexadecimal value.

1200 • **integer**

1201 All numeric values are 32-bit signed integers.

1202 • **ReadReport**

1203 A read report is returned by the `rawReadIDs` and `readIDs` commands, and is also

1204 included in notifications. The data included depends on the settings of the associated
1205 `DataSelector`. The formatting is MTB-specific.

- 1206 • **string**
1207 All strings SHALL be capable of being represented with UTF-8 encoding.
- 1208 • **timestamp**
1209 A timestamp (date and time) with millisecond accuracy. The exact format depends on
1210 the MTB in use.
- 1211 • **triggervalue**
1212 A trigger definition, formatted as a string. The legal values depend on the type of the
1213 trigger.
- 1214 • **type[]**
1215 A list of the specified data type. The list is unordered unless explicitly stated
1216 otherwise.
- 1217 • **void**
1218 No data, indicating no parameters or return value (depending on where it's used).
1219

1220 In addition, many commands have object types like `DataSelector`, `TagSelector`
1221 etc. as parameter or return value. The MTB defines how objects are encoded in the
1222 command and reply messages.

6.1 Object ReaderDevice

The ReaderDevice object represents the Reader. It is a singleton object that SHALL exist by default. There are therefore no commands to create or delete it.

The ReaderDevice object keeps lists of all Sources, DataSelectors, NotificationChannels, Triggers and TagSelectors known. The objects SHALL be added to these lists implicitly when created.

6.1.1 ReaderDevice.getEPC

Returns the EPC code of the Reader. Since the EPC is set by the manufacturer, there is no setEPC() command.

Compliance Requirement: Compliant systems SHALL implement this command.

Usage:

ReaderDevice.getEPC (void): epc

Parameter(s):

Data Type: void . This command SHALL NOT take any parameters.

Return Value(s):

Data Type: epc . The EPC code of the Reader.

6.1.2 ReaderDevice.getManufacturer

Returns the name of the Reader manufacturer.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

ReaderDevice.getManufacturer (void): string

Parameter(s):

1256 Data Type: `void`. This command SHALL NOT take any parameters.

1257

1258 **Return Value(s):**

1259 Data Type: `string`. The name of the manufacturer of the Reader.

1260

1261

1262

1263 **6.1.3 ReaderDevice.getModel**

1264 Returns the manufacturer-defined model description of the Reader.

1265

1266 **Compliance Requirement:** Compliant systems MAY implement this command.

1267

1268 **Usage:**

1269 `ReaderDevice.getModel (void): string`

1270

1271 **Parameter(s):**

1272 Data Type: `void`. This command SHALL NOT take any parameters.

1273

1274 **Return Value(s):**

1275 Data Type: `string`. The name of the Reader model, defined by the manufacturer.

1276

1277

1278

1279 **6.1.4 ReaderDevice.getHandle**

1280 Returns the Reader handle. The reader handle is an arbitrary integer number, settable by
1281 the Host, to distinguish between different readers. The handle can be used as an
1282 alternative to the Reader name, however, it is more code-friendly than a name and
1283 therefore intended for use by applications.

1284

1285 **Compliance Requirement:** Compliant systems MAY implement this command.

1286

1287 **Usage:**

1288 `ReaderDevice.getHandle(void): integer`

1289

1290 **Parameter(s):**

1291 Data Type: `void`. This command SHALL NOT take any parameters.

1292

1293 **Return Value(s):**

1294 Data Type: `integer`. The handle of the Reader. If no handle has been set, 0 (zero)
1295 SHALL be returned.

1296

1297

1298

1299 **6.1.5 ReaderDevice.setHandle**

1300 Sets the Reader's handle.

1301

1302 **Compliance Requirement:** Compliant systems MAY implement this command.

1303

1304 **Usage:**

1305 `ReaderDevice.setHandle (handle: integer): void`

1306

1307 **Parameter(s):**

1308 `handle` – Data Type: `integer`. A handle to assign to the Reader.

1309

1310 **Return Value(s):**

1311 Data Type: `void`. This command SHALL NOT return any value.

1312

1313

1314

1315 **6.1.6 ReaderDevice.getName**

1316 Returns the name of the Reader. The reader name is an arbitrary string, settable by the
1317 Host, to distinguish between different readers. The name can be used as an alternative to
1318 the Reader handle, however, it is intended to be easier understandable for the human user
1319 and therefore used for display in user interfaces.

1320

1321 **Compliance Requirement:** Compliant systems SHALL implement this command.
1322
1323 **Usage:**
1324 ReaderDevice.getName (void): string
1325
1326 **Parameter(s):**
1327 Data Type: void . This command SHALL NOT take any parameters.
1328
1329 **Return Value(s):**
1330 Data Type: string . The name of the reader. If no name has been set, an empty string
1331 SHALL be returned.
1332
1333
1334

1335 **6.1.7 ReaderDevice.setName**

1336 Sets the name of the Reader.

1337
1338 **Compliance Requirement:** Compliant systems SHALL implement this command.
1339

1340 **Usage:**
1341 ReaderDevice.setName (name: string): void
1342

1343 **Parameter(s):**
1344 name – Data Type: string . A name to assign to the Reader.
1345

1346 **Return Value(s):**
1347 Data Type: void . This command SHALL NOT return any value.
1348

1349
1350

6.1.8 ReaderDevice.getRole

Returns the Reader's role. The role is an arbitrary string, settable by the Host, to identify the task or function of a reader, e.g. that this reader is at a receiving door.

Compliance Requirement: Compliant systems SHALL implement this command.

Usage:

```
ReaderDevice.getRole (void): string
```

Parameter(s):

Data Type: void. This command SHALL NOT take any parameters.

Return Value(s):

Data Type: string. The role of the reader.

6.1.9 ReaderDevice.setRole

The Host sets the Readers user-defined functional name or purpose.

Compliance Requirement: Compliant systems SHALL implement this command.

Usage:

```
ReaderDevice.setRole (role: string): void
```

Parameter(s):

role – Data Type: string. A role to assign to the Reader.

Return Value(s):

Data Type: void. This command SHALL NOT return any value.

1383

1384 **6.1.10 ReaderDevice.getTimeTicks**

1385 Returns the current “uptime ticks” of a Reader. A compliant Reader SHALL have an
1386 internal clock that is reset when booted. The uptime ticks are the number of clock ticks
1387 since the reader was booted last.

1388

1389 **Compliance Requirement:** Compliant systems SHALL implement this command.

1390

1391 **Usage:**

1392 `ReaderDevice.getTimeTicks (void): integer`

1393

1394 **Parameter(s):**

1395 Data Type: `void`. This command SHALL NOT take any parameters.

1396

1397 **Return Value(s):**

1398 Data Type: `integer`. The current uptime time ticks of the Reader.

1399

1400

1401

1402 **6.1.11 ReaderDevice.getTimeUTC**

1403 Returns the current absolute (UTC) time of the Reader.

1404

1405 **Compliance Requirement:** Compliant systems MAY implement this command.

1406

1407 **Usage:**

1408 `ReaderDevice.getTimeUTC (void): timestamp`

1409

1410 **Parameter(s):**

1411 Data Type: `void`. This command SHALL NOT take any parameters.

1412

1413 **Return Value(s):**

1414 Data Type: `timestamp`. The current absolute (UTC) time.

1415
1416
1417

1418 **6.1.12 ReaderDevice.setTimeUTC**

1419 Sets the Readers absolute (UTC) time.

1420

1421 **Compliance Requirement:** Compliant systems MAY implement this command.

1422

1423 **Usage:**

1424 `ReaderDevice.setTimeUTC (utc: timestamp): void`

1425

1426 **Parameter(s):**

1427 `utc` – Data Type: `timestamp`. The absolute (UTC) time to assign to the Reader.

1428

1429 **Return Value(s):**

1430 Data Type: `void`. This command SHALL NOT return any value.

1431

1432

1433

1434 **6.1.13 ReaderDevice.getManufacturerDescription**

1435 Returns a description of the Reader provided by the manufacturer. This MAY include
1436 information about the firmware version used and other information deemed important by
1437 the manufacturer. The format is implementation-dependent and SHALL be described in
1438 the reader documentation.

1439

1440 **Compliance Requirement:** Compliant systems MAY implement this command.

1441

1442 **Usage:**

1443 `ReaderDevice.getManufacturerDescription (void): string`

1444

1445 **Parameter(s):**

1446 Data Type: `void`. This command SHALL NOT take any parameters.

1447

1448 **Return Value(s):**

1449 Data Type: `string`. The reader description, as set by the manufacturer.

1450

1451

1452

1453 **6.1.14 ReaderDevice.getCurrentSource**

1454 Returns the current `Source` of the Reader. If the Reader supports one or more
1455 `Sources` then the Reader SHALL provide a Current Source. The Current Source MAY
1456 be pre-configured by the Reader manufacturer or the Reader MAY support commands
1457 that allow setting the Current Source. If the Reader supports only one source, then the
1458 Current Source SHALL be a constant equated to the one source.

1459

1460 **Compliance Requirement:** Compliant systems SHALL implement this command.

1461

1462 **Usage:**

1463 `ReaderDevice.getCurrentSource (void): Source`

1464

1465 **Parameter(s):**

1466 Data Type: `void`. This command SHALL NOT take any parameters.

1467

1468 **Return Value(s):**

1469 Data Type: `Source`. The `Source` object currently set to be the current Source.

1470

1471

1472

1473 **6.1.15 ReaderDevice.setCurrentSource**

1474 Sets the current `Source` object.

1475

1476 **Compliance Requirement:** Compliant systems MAY implement this command.

1477

1478 **Usage:**

1479 `ReaderDevice.setCurrentSource (currentSource: Source): void`

1480

1481 **Parameter(s):**
1482 `currentSource` – Data Type: `Source`. The `Source` to be assigned as current
1483 `Source`.

1484
1485 **Return Value(s):**
1486 Data Type: `void`. This command SHALL NOT return any value.

1487

1488

1489

1490 **6.1.16 ReaderDevice.getCurrentDataSelector**

1491 Returns the current `DataSelector` of the Reader. If *null*, a new `DataSelector` is
1492 created with the name “ReaderProtocol:Default”, initialized with `FieldNames` =
1493 “AllSupported”, and `EventFilters` = “AllSupported”, and assigned to the current
1494 `DataSelector` prior to returning the result, equivalent to the following pseudo-code:

1495

```
1496 DataSelector ReaderDevice.getCurrentDataSelector()  
1497 {  
1498     if currentDS == null  
1499     {  
1500         ds = DataSelector.create("ReaderProtocol:Default")  
1501         ds.addFieldNames([AllSupported])  
1502         ds.addEventFilters([AllSupported])  
1503         currentDS = ds  
1504     }  
1505     return currentDS  
1506 }
```

1507

1508 **Compliance Requirement:** Compliant systems SHALL implement this command.

1509

1510 **Usage:**

1511 `ReaderDevice.getCurrentDataSelector (void): DataSelector`

1512

1513 **Parameter(s):**

1514 Data Type: `void`. This command SHALL NOT take any parameters.

1515

1516 **Return Value(s):**

1517 Data Type: `DataSelector`. The `DataSelector` object currently set to be the
1518 current `DataSelector`.

1519

1520

1521

1522 **6.1.17 ReaderDevice.setCurrentDataSelector**

1523 Sets the current `DataSelector` object.

1524

1525 **Compliance Requirement:** Compliant systems MAY implement this command.

1526

1527 **Usage:**

```
1528 ReaderDevice.setCurrentDataSelector (  
1529     currentDataSelector: DataSelector): void
```

1530

1531 **Parameter(s):**

1532 `currentDataSelector` – Data Type: `DataSelector`. The `DataSelector` to
1533 be assigned as current `DataSelector`.

1534

1535 **Return Value(s):**

1536 Data Type: `void`. This command SHALL NOT return any value.

1537

1538

1539

1540 **6.1.18 ReaderDevice.removeSources**

1541 Removes the specified `Sources` from the list of `Sources` currently associated with
1542 this Reader. If one or more of the `Sources` given are not known, or if some of the
1543 `Sources` to be removed are currently not associated with this Reader, these are ignored
1544 and all other `Sources` SHALL be removed and the command SHALL complete
1545 successfully.

1546 However, if one or more `Sources` of the list is a fixed `Source`, the command SHALL
1547 return in an error and no `Sources` SHALL be removed.

1548 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1549 (`ERROR_*_NOT_FOUND`).

1550

1551 **Compliance Requirement:** Compliant systems MAY implement this command.

1552

1553 **Usage:**

1554 `ReaderDevice.removeSources (sources: Source[]): void`

1555

1556 **Parameter(s):**

1557 `sources` – Data Type: `Source[]`. The list of `Sources` to be removed.

1558

1559 **Return Value(s):**

1560 Data Type: `void`. This command SHALL NOT return a value.

1561

1562

1563

1564 **6.1.19 ReaderDevice.removeAllSources**

1565 Removes all `Sources` currently associated with this Reader, except the ones marked as
1566 fixed.

1567 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1568 (`ERROR_*_NOT_FOUND`).

1569

1570 **Compliance Requirement:** Compliant systems MAY implement this command.

1571

1572 **Usage:**

1573 `ReaderDevice.removeAllSources (void): void`

1574

1575 **Parameter(s):**

1576 Data Type: `void`. This command SHALL NOT take any parameters.

1577

1578 **Return Value(s):**

1579 Data Type: `void`. This command SHALL NOT return any value.

1580

1581

1582

6.1.20 ReaderDevice.getSource

Returns the Source with the specified. If no Source object with the given name exists, the error ERROR_SOURCE_NOT_FOUND is raised.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

ReaderDevice.getSource (name: string): Source

Parameter(s):

name – Data Type: string . The name of the Source object to be returned.

Return Value(s):

Data Type: Source . The requested Source object.

6.1.21 ReaderDevice.getAllSources

Returns all Sources currently associated with this Reader. If no Sources are currently associated with the Reader, the command SHALL complete successfully and an empty list SHALL be returned.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

ReaderDevice.getAllSources (void): Source[]

Parameter(s):

Data Type: void . This command SHALL NOT take any parameters.

Return Value(s):

Data Type: Source[] . The list of all Source objects currently associated with the Reader.

1616

1617

1618

1619 **6.1.22 ReaderDevice.removeDataSelectors**

1620 Removes the specified `DataSelectors` from the list of `DataSelectors` currently
1621 associated with this Reader. If one or more of the `DataSelectors` given are not
1622 known, or if some of the `DataSelectors` to be removed are currently not associated
1623 with this Reader, these are ignored and all other `DataSelectors` SHALL be removed
1624 and the command SHALL complete successfully.

1625 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1626 (`ERROR_*_NOT_FOUND`).

1627

1628 **Compliance Requirement:** Compliant systems MAY implement this command.

1629

1630 **Usage:**

```
1631 ReaderDevice.removeDataSelectors(  
1632     dataSelectors: DataSelector[]): void
```

1633

1634 **Parameter(s):**

1635 `dataSelectors` – Data Type: `DataSelector[]`. The list of `DataSelectors`
1636 to be removed.

1637

1638 **Return Value(s):**

1639 Data Type: `void`. This command SHALL NOT return any value.

1640

1641

1642

1643 **6.1.23 ReaderDevice.removeAllDataSelectors**

1644 Removes all `DataSelectors` currently associated with this Reader.

1645 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1646 (`ERROR_*_NOT_FOUND`).

1647

1648 **Compliance Requirement:** Compliant systems MAY implement this command.

1649

1650 **Usage:**
1651 `ReaderDevice.removeAllDataSelectors (void): void`

1652
1653 **Parameter(s):**
1654 Data Type: `void`. This command SHALL NOT take any parameters.

1655
1656 **Return Value(s):**
1657 Data Type: `void`. This command SHALL NOT return any value.

1658
1659
1660

1661 **6.1.24 ReaderDevice.getDataSelector**

1662 Returns the `DataSelector` with the specified name currently associated with this
1663 Reader. If no `DataSelector` object with the given name exists, the error
1664 `ERROR_DATASELECTOR_NOT_FOUND` is raised.

1665
1666 **Compliance Requirement:** Compliant systems MAY implement this command.

1667
1668 **Usage:**
1669 `ReaderDevice.getDataSelector (name: string): DataSelector`

1670
1671 **Parameter(s):**
1672 `name` – Data Type: `string`. The name of the `DataSelector` object to be returned.

1673
1674 **Return Value(s):**
1675 Data Type: `DataSelector`. The requested `DataSelector` object.

1676
1677
1678

1679 **6.1.25 ReaderDevice.getAllDataSelectors**

1680 Returns all `DataSelectors` currently associated with the Reader. If no
1681 `DataSelectors` are currently associated with this object, the command SHALL
1682 complete successfully and an empty list SHALL be returned.

1683

1684 **Compliance Requirement:** Compliant systems MAY implement this command.

1685

1686 **Usage:**

1687 `ReaderDevice.getAllDataSelectors (void): DataSelector[]`

1688

1689 **Parameter(s):**

1690 Data Type: `void`. This command SHALL NOT take any parameters.

1691

1692 **Return Value(s):**

1693 Data Type: `DataSelector[]`. The list of all `DataSelector` objects currently
1694 associated with the Reader.

1695

1696

1697

1698 **6.1.26 ReaderDevice.removeNotificationChannels**

1699 Removes the specified `NotificationChannels` from the list of
1700 `NotificationChannels` currently associated with the Reader. If one or more of the
1701 `NotificationChannels` given are not known, or if some of the
1702 `NotificationChannels` to be removed are currently not associated with the
1703 Reader, these are ignored and all other `NotificationChannels` SHALL be
1704 removed and the command SHALL complete successfully.

1705 Removing a 'default' object changes the 'default' to *null*. Using a 'null' object is an error
1706 (`ERROR_*_NOT_FOUND`).

1707

1708 **Compliance Requirement:** Compliant systems MAY implement this command.

1709

1710 **Usage:**

1711 `ReaderDevice.removeNotificationChannels (`
1712 `channels: NotificationChannel[]): void`

1713

1714 **Parameter(s):**

1715 `channels` – Data Type: `NotificationChannel[]`. The list of
1716 `NotificationChannels` to be removed.

1717

1718 **Return Value(s):**

1719 Data Type: `void`. This command SHALL NOT return any value.

1720

1721

1722

1723 **6.1.27 ReaderDevice.removeAllNotificationChannels**

1724 Removes all `NotificationChannels` currently associated with the Reader.

1725 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1726 (`ERROR_*_NOT_FOUND`).

1727

1728 **Compliance Requirement:** Compliant systems MAY implement this command.

1729

1730 **Usage:**

1731 `ReaderDevice.removeAllNotificationChannels (void): void`

1732

1733 **Parameter(s):**

1734 Data Type: `void`. This command SHALL NOT take any parameters.

1735

1736 **Return Value(s):**

1737 Data Type: `void`. This command SHALL NOT return any value.

1738

1739

1740

1741 **6.1.28 ReaderDevice.getNotificationChannel**

1742 Returns the `NotificationChannel` with the specified name currently associated
1743 with the Reader. If no `NotificationChannel` object with the given name exists, the
1744 error `ERROR_CHANNEL_NOT_FOUND` is raised.

1745

1746 **Compliance Requirement:** Compliant systems MAY implement this command.

1747

1748 **Usage:**

1749 `ReaderDevice.getNotificationChannel (`
1750 `name: String): NotificationChannel`

1751

1752 **Parameter(s):**

1753 name – Data Type: `string`. The name of the `NotificationChannel` object to
1754 be returned.

1755

1756 **Return Value(s):**

1757 Data Type: `NotificationChannel`. The requested `NotificationChannel`
1758 object.

1759

1760

1761

1762 **6.1.29 ReaderDevice.getAllNotificationChannels**

1763 Returns all `NotificationChannels` currently associated with the Reader. If no
1764 `NotificationChannels` are currently associated with this object, the command
1765 SHALL complete successfully and an empty list SHALL be returned.

1766

1767 **Compliance Requirement:** Compliant systems MAY implement this command.

1768

1769 **Usage:**

1770 `ReaderDevice.getAllNotificationChannels (void):`
1771 `NotificationChannel[]`

1772

1773 **Parameter(s):**

1774 Data Type: `void`. This command SHALL NOT take any parameters.

1775

1776 **Return Value(s):**

1777 Data Type: `NotificationChannel[]`. The list of all `NotificationChannel`
1778 objects currently associated with the Reader.

1779

1780

1781

1782 **6.1.30 ReaderDevice.removeTriggers**

1783 Removes the specified `Triggers` from the list of `Triggers` currently associated with
1784 the Reader. If one or more of the `Triggers` given are not known, or if some of the

1785 Triggers to be removed are currently not associated with the Reader, these are ignored
1786 and all other Triggers SHALL be removed and the command SHALL complete
1787 successfully.

1788 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1789 (ERROR_*_NOT_FOUND).

1790

1791 **Compliance Requirement:** Compliant systems MAY implement this command.

1792

1793 **Usage:**

1794 ReaderDevice.removeTriggers (
1795 triggers: Trigger[]): void

1796

1797 **Parameter(s):**

1798 triggers – Data Type: Trigger[]. The list of Triggers to be removed.

1799

1800 **Return Value(s):**

1801 Data Type: void. This command SHALL NOT return any value.

1802

1803

1804

1805 **6.1.31 ReaderDevice.removeAllTriggers**

1806 Removes all Triggers currently associated with the Reader.

1807 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1808 (ERROR_*_NOT_FOUND).

1809

1810 **Compliance Requirement:** Compliant systems MAY implement this command.

1811

1812 **Usage:**

1813 ReaderDevice.removeAllTriggers (void): void

1814

1815 **Parameter(s):**

1816 Data Type: void. This command SHALL NOT take any parameters.

1817

1818 **Return Value(s):**

1819 Data Type: `void`. This command SHALL NOT return any value.

1820

1821

1822

1823 **6.1.32 ReaderDevice.getTrigger**

1824 Returns the `Trigger` with the specified name currently associated with the Reader. If

1825 no `Trigger` object with the given name exists, the error

1826 `ERROR_TRIGGER_NOT_FOUND` is raised.

1827

1828 **Compliance Requirement:** Compliant systems MAY implement this command.

1829

1830 **Usage:**

```
1831 ReaderDevice.getTrigger (  
1832     name: String): Trigger
```

1833

1834 **Parameter(s):**

1835 name – Data Type: `string`. The name of the `Trigger` object to be returned.

1836

1837 **Return Value(s):**

1838 Data Type: `Trigger`. The requested `Trigger` object.

1839

1840

1841

1842 **6.1.33 ReaderDevice.getAllTriggers**

1843 Returns all `Triggers` currently associated with the Reader. If no `Triggers` are

1844 currently associated with this object, the command SHALL complete successfully and an

1845 empty list SHALL be returned.

1846

1847 **Compliance Requirement:** Compliant systems MAY implement this command.

1848

1849 **Usage:**

```
1850 ReaderDevice.getAllTriggers (void): Trigger[]
```

1851

1852 **Parameter(s):**

1853 Data Type: `void`. This command SHALL NOT take any parameters.

1854

1855 **Return Value(s):**

1856 Data Type: `Trigger[]`. The list of all `Trigger` objects currently associated with the
1857 Reader.

1858

1859

1860

1861 **6.1.34 ReaderDevice.removeTagSelectors**

1862 Removes the specified `TagSelectors` from the list of `TagSelectors` currently
1863 associated with the Reader. If one or more of the `TagSelectors` given are not known,
1864 or if some of the `TagSelectors` to be removed are currently not associated with the
1865 Reader, these are ignored and all other `TagSelectors` SHALL be removed and the
1866 command SHALL complete successfully.

1867 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1868 (`ERROR_*_NOT_FOUND`).

1869

1870 **Compliance Requirement:** Compliant systems MAY implement this command.

1871

1872 **Usage:**

1873 `ReaderDevice.removeTagSelectors (`
1874 `selectors: TagSelector[]): void`

1875

1876 **Parameter(s):**

1877 `selectors` – Data Type: `TagSelector[]`. The list of `TagSelectors` to be
1878 removed.

1879

1880 **Return Value(s):**

1881 Data Type: `void`. This command SHALL NOT return any value.

1882

1883

1884

6.1.35 ReaderDevice.removeAllTagSelectors

Removes all TagSelectors currently associated with the Reader.

Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error (ERROR_*_NOT_FOUND).

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
ReaderDevice.removeAllTagSelectors (void): void
```

Parameter(s):

Data Type: void. This command SHALL NOT take any parameters.

Return Value(s):

Data Type: void. This command SHALL NOT return any value.

6.1.36 ReaderDevice.getTagSelector

Returns the TagSelector with the specified name currently associated with the Reader. If no TagSelector object with the given name exists, the error ERROR_FILTER_NOT_FOUND is raised.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
ReaderDevice.getTagSelector (  
    name: String): TagSelector
```

Parameter(s):

name – Data Type: string. The name of the TagSelector object to be returned.

Return Value(s):

1918 Data Type: `TagSelector` . The requested `TagSelector` object.

1919

1920

1921

1922 **6.1.37 ReaderDevice.getAllTagSelectors**

1923 Returns all `TagSelectors` currently associated with the Reader. If no
1924 `TagSelectors` are currently associated with this object, the command SHALL
1925 complete successfully and an empty list SHALL be returned.

1926

1927 **Compliance Requirement:** Compliant systems MAY implement this command.

1928

1929 **Usage:**

1930 `ReaderDevice.getAllTagSelectors (void): TagSelector[]`

1931

1932 **Parameter(s):**

1933 Data Type: `void` . This command SHALL NOT take any parameters.

1934

1935 **Return Value(s):**

1936 Data Type: `TagSelector[]` . The list of all `TagSelector` objects currently
1937 associated with the Reader.

1938

1939

1940

1941 **6.1.38 ReaderDevice.removeTagFields**

1942 Removes the specified `TagFields` from the list of `TagFields` currently associated
1943 with the Reader. If one or more of the `TagFields` given are not known, or if some of
1944 the `TagFields` to be removed are currently not associated with the Reader, these are
1945 ignored and all other `TagFields` SHALL be removed and the command SHALL
1946 complete successfully.

1947 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1948 (`ERROR_*_NOT_FOUND`).

1949

1950 **Compliance Requirement:** Compliant systems MAY implement this command.

1951

1952 **Usage:**

1953 `ReaderDevice.removeTagFields (`
1954 `fields: TagField[]): void`

1955

1956 **Parameter(s):**

1957 `fields` – Data Type: `TagField[]`. The list of `TagFields` to be removed.

1958

1959 **Return Value(s):**

1960 Data Type: `void`. This command SHALL NOT return any value.

1961

1962

1963

1964 **6.1.39 ReaderDevice.removeAllTagFields**

1965 Removes all `TagFields` currently associated with the Reader.

1966 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
1967 (`ERROR_*_NOT_FOUND`).

1968

1969 **Compliance Requirement:** Compliant systems MAY implement this command.

1970

1971 **Usage:**

1972 `ReaderDevice.removeAllTagFields (void): void`

1973

1974 **Parameter(s):**

1975 Data Type: `void`. This command SHALL NOT take any parameters.

1976

1977 **Return Value(s):**

1978 Data Type: `void`. This command SHALL NOT return any value.

1979

1980

1981

6.1.40 ReaderDevice.getTagField

Returns the TagField with the specified name currently associated with the Reader. If no TagField object with the given name exists, the error ERROR_TAGFIELD_NOT_FOUND is raised.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
ReaderDevice.getTagField (  
    name: String): TagField
```

Parameter(s):

name – Data Type: string. The name of the TagField object to be returned.

Return Value(s):

Data Type: TagField. The requested TagField object.

6.1.41 ReaderDevice.getAllTagFields

Returns all TagFields currently associated with the Reader. If no TagFields are currently associated with this object, the command SHALL complete successfully and an empty list SHALL be returned.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
ReaderDevice.getAllTagFields (void): TagField[]
```

Parameter(s):

Data Type: void. This command SHALL NOT take any parameters.

Return Value(s):

2015 Data Type: `TagField[]`. The list of all `TagField` objects currently associated with
2016 the Reader.

2017

2018

2019

2020 **6.1.42 ReaderDevice.resetToDefaultSettings**

2021 Resets all internal state variables of the Reader to a default configuration. The
2022 documentation of the Reader SHALL provide a definition what the default settings are.

2023

2024 **Compliance Requirement:** Compliant systems SHALL implement this command.

2025

2026 **Usage:**

2027 `ReaderDevice.resetToDefaultSettings (void): void`

2028

2029 **Parameter(s):**

2030 Data Type: `void`. This command SHALL NOT take any parameters.

2031

2032 **Return Value(s):**

2033 Data Type: `void`. This command SHALL NOT return any value.

2034

2035

2036

2037 **6.1.43 ReaderDevice.reboot**

2038 The Host asks the Reader to stop any pending data acquisition, reinitialize all variables to
2039 its power up state and execute any initialization functions it normally performs on power
2040 up.

2041 This command SHALL behave as follows:

2042 1. Check if a reboot is possible. If so the command SHALL return without an error,
2043 else an error is returned.

2044 2. Send a `goodbye` command on all notification channels.

2045 3. Execute the reboot

2046 It is then the responsibility of the host to reestablish the communication channels.

2047

2048 **Compliance Requirement:** Compliant systems SHALL implement this command.

2049

2050 **Usage:**

2051 ReaderDevice.reboot (void): void

2052

2053 **Parameter(s):**

2054 Data Type: void. This command SHALL NOT take any parameters.

2055

2056 **Return Value(s):**

2057 Data Type: void. This command SHALL NOT return any value.

2058

2059

2060

2061 **6.1.44 ReaderDevice.goodbye**

2062 This command is used to indicate a ‘graceful disconnect’ distinguishing proper system
2063 operation from a ‘disgraceful disconnect’ due to loss of network connectivity, unexpected
2064 shutdown of a Host or Reader, or other ‘surprising’ conditions. Upon receiving the reply
2065 to this command, the connection on the control channel will be terminated by the Host.

2066

2067 **Compliance Requirement:** Compliant systems SHALL implement this command.

2068

2069 **Usage:**

2070 ReaderDevice.goodbye (void): void

2071

2072 **Parameter(s):**

2073 Data Type: void. This command SHALL NOT take any parameters.

2074

2075 **Return Value(s):**

2076 Data Type: void. This command SHALL NOT return any value.

2077

2078

2079

2080 **6.1.45 ReaderDevice.getReadPoint**

2081 Returns the `ReadPoint` with the specified name currently associated with the Reader. If
2082 no `ReadPoint` object with the given name exists, the error
2083 `ERROR_READPOINT_NOT_FOUND` is raised.

2085 **Compliance Requirement:** Compliant systems MAY implement this command.

2087 **Usage:**

```
2088 ReaderDevice.getReadPoint (  
2089     name: String): ReadPoint
```

2091 **Parameter(s):**

2092 name – Data Type: `string`. The name of the `ReadPoint` object to be returned.

2094 **Return Value(s):**

2095 Data Type: `ReadPoint`. The requested `ReadPoint` object.

2099 **6.1.46 ReaderDevice.getAllReadPoints**

2100 Returns all `ReadPoints` of the Reader. This list is preconfigured by the manufacturer.

2102 **Compliance Requirement:** Compliant systems MAY implement this command.

2104 **Usage:**

```
2105 ReaderDevice.getAllReadPoints (void): ReadPoint[]
```

2107 **Parameter(s):**

2108 Data Type: `void`. This command SHALL NOT take any parameters.

2110 **Return Value(s):**

2111 Data Type: `ReadPoint[]`. The list of all `ReadPoints`.

2113

2114

2115 **6.2 Object Source**

2116 **6.2.1 Source.create**

2117 Create a **Source** object with a given name. If a **Source** object with the same name
2118 exists already, an error is returned. This is a static method.

2119 The **Source** SHALL implicitly be added to the list of all **Sources** kept by the
2120 **ReaderDevice** object.

2121 The newly created object has its standard properties (if supported) initialized as:

Property	Value
Name	name
Fixed	<instance/vendor dependent>
ReadPoints	[]
ReadTriggers	[]
TagSelectors	[]
GlimpsedTimeout	<instance/vendor dependent>
ObservedThreshold	<instance/vendor dependent>
ObservedTimeout	<instance/vendor dependent>
LostTimeout	<instance/vendor dependent>
ReadCyclesPerTrigger	<instance/vendor dependent>
MaxDutyReadCycle	<instance/vendor dependent>
ReadTimeout	<instance/vendor dependent>
Session	<instance/vendor dependent>

2122 **Table 6: Default properties for Source object**

2123

2124 **Compliance Requirement:** Compliant systems MAY implement this command.

2125

2126 **Usage:**

2127 `Source.create(name: string): Source`

2128

2129 **Parameter(s):**

2130 `name` – Data Type: `string`. The name of the object to be created.

2131

2132 **Return Value(s):**

2133 Data Type: `Source`. The newly created object.

2134

2135

2136

2137 **6.2.2 Source.getName**

2138 Returns the name of the given `Source` object.

2139

2140 **Compliance Requirement:** Compliant systems MAY implement this command.

2141

2142 **Usage:**

2143 `Source.getName(void): string`

2144

2145 **Parameter(s):**

2146 Data Type: `void`. This command SHALL NOT take any parameters.

2147

2148 **Return Value(s):**

2149 Data Type: `string`. The name of the `Source` object is returned.

2150

2151

2152

2153

2154 **6.2.3 Source.isFixed**

2155 This command can be used to find out if a specific `Source` is fixed or if it can be
2156 changed, i.e. if `ReadPoints` can be added or removed (see section 5.2).

2157

2158 **Compliance Requirement:** Compliant systems MAY implement this command.

2159

2160 **Usage:**

2161 `Source.isFixed(void): boolean`

2162

2163 **Parameter(s):**

2164 Data Type: void . This command SHALL NOT take any parameters.

2165

2166 **Return Value(s):**

2167 Data Type: boolean . “true”, if the Source is fixed, “false” otherwise.

2168

2169

2170

2171 **6.2.4 Source.addReadPoints**

2172 Adds the specified ReadPoints to the list of Readpoints currently associated with
2173 this Source. If one or more of the ReadPoints given are not known, an error SHALL
2174 be raised and no ReadPoints SHALL be added. If some of the ReadPoints to be
2175 added are already associated with this Source, only the not yet associated
2176 ReadPoints shall be added and the command SHALL complete successfully.

2177

2178 **Compliance Requirement:** Compliant systems MAY implement this command.

2179

2180 **Usage:**

2181 Source.addReadPoints(readPoints: ReadPoint[]): void

2182

2183 **Parameter(s):**

2184 readPoints - Data Type: ReadPoint[] . The list of ReadPoints to be added. If
2185 the Reader implementation does not allow ReadPoints to be shared between
2186 Sources and any of the specified ReadPoints are already assigned to another
2187 Source, then this command SHALL report the error ERROR_READPOINT_IN_USE
2188 and no ReadPoints will be added to the Source.

2189

2190 **Return Value(s):**

2191 Data Type: void . This command SHALL NOT return any value.

2192

2193

2194

6.2.5 Source.removeReadPoints

Removes the specified ReadPoints from the list of ReadPoints currently associated with this Source. If one or more of the ReadPoints given are not known, or if some of the ReadPoints to be removed are currently not associated with this Source, these are ignored and all other ReadPoints SHALL be removed and the command SHALL complete successfully.

Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error (ERROR_*_NOT_FOUND).

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
Source.removeReadPoints(readPoints: ReadPoint[]): void
```

Parameter(s):

readPoints – Data Type: ReadPoint[]. The list of ReadPoints to be removed.

Return Value(s):

Data Type: void. This command SHALL NOT return any value.

6.2.6 Source.removeAllReadPoints

Removes all ReadPoints currently associated with this Source.

Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error (ERROR_*_NOT_FOUND).

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
Source.removeAllReadPoints(void):void
```

2229 **Parameter(s):**

2230 Data Type: `void`. This command SHALL NOT take any parameters.

2231

2232 **Return Value(s):**

2233 Data Type: `void`. This command SHALL NOT return any value.

2234

2235

2236

2237

2238 **6.2.7 Source.getReadPoint**

2239 Returns the `ReadPoint` with the specified name currently associated with this `Source`
2240 object. If no `ReadPoint` object with the given name exists, the error
2241 `ERROR_READPOINT_NOT_FOUND` is raised.

2242

2243 **Compliance Requirement:** Compliant systems MAY implement this command.

2244

2245 **Usage:**

2246 `Source.getReadPoint(name: string): ReadPoint`

2247

2248 **Parameter(s):**

2249 name – Data Type: `string`. The name of the `ReadPoint` object to be returned.

2250

2251 **Return Value(s):**

2252 Data Type: `ReadPoint`. The requested `ReadPoint` object.

2253

2254

2255

2256

2257 **6.2.8 Source.getAllReadPoints**

2258 Returns all `ReadPoints` currently associated with this `Source` object. If no
2259 `ReadPoints` are currently associated with this object, the command SHALL complete
2260 successfully and an empty list SHALL be returned.

2261

2262 **Compliance Requirement:** Compliant systems MAY implement this command.
2263
2264 **Usage:**
2265 Source.getAllReadPoints(void): ReadPoint[]
2266
2267 **Parameter(s):**
2268 Data Type: void. This command SHALL NOT take any parameters.
2269
2270 **Return Value(s):**
2271 Data Type: ReadPoint[] . The list of all ReadPoint objects currently associated
2272 with this Source.
2273
2274
2275
2276

2277 **6.2.9 Source.addReadTriggers**

2278 Adds the specified Triggers to the list of Read Triggers currently associated with this
2279 Source. If one or more of the Triggers given are not supported or are not known, an
2280 error SHALL be raised and no Triggers SHALL be added. If some of the Triggers
2281 to be added are already associated with this Source, only the not yet associated
2282 Triggers shall be added and the command SHALL complete successfully.
2283 Once a Trigger has been added, it is immediately activated.
2284

2285 **Compliance Requirement:** Compliant systems MAY implement this command.

2286
2287 **Usage:**
2288 Source.addReadTriggers(triggers: Trigger[]): void
2289

2290 **Parameter(s):**
2291 triggers – Data Type: Trigger[] . The list of Triggers to be added.
2292

2293 **Return Value(s):**
2294 Data Type: void. This command SHALL NOT return any value.

2295
2296
2297
2298

2299 **6.2.10 Source.removeReadTriggers**

2300 Removes the specified `Triggers` from the list of Read Triggers currently associated
2301 with this `Source`. If one or more of the `Triggers` given are not supported or not
2302 known, or if some of the `Triggers` to be removed are currently not associated with this
2303 `Source`, these are ignored and all other `Triggers` SHALL be removed and the
2304 command SHALL complete successfully.

2305 Once a Trigger isn't associated to any object anymore, it is deactivated.

2306 Removing a 'default' object changes the 'default' to `null`. Using a '`null`' object is an error
2307 (`ERROR_*_NOT_FOUND`).

2308

2309 **Compliance Requirement:** Compliant systems MAY implement this command.

2310

2311 **Usage:**

2312 `Source.removeReadTriggers(triggers:Trigger[]):void`

2313

2314 **Parameter(s):**

2315 `triggers` – Data Type: `Trigger[]`. The list of `Triggers` to be removed.

2316

2317 **Return Value(s):**

2318 Data Type: `void`. This command SHALL NOT return any value.

2319

2320

2321

2322

2323 **6.2.11 Source.removeAllReadTriggers**

2324 Removes all `Triggers` currently associated with this `Source`.

2325 Removing a 'default' object changes the 'default' to `null`. Using a '`null`' object is an error
2326 (`ERROR_*_NOT_FOUND`).

2327

2328 **Compliance Requirement:** Compliant systems MAY implement this command.

2329

2330 **Usage:**

2331 `Source.removeAllReadTriggers(void): void`

2332

2333 **Parameter(s):**

2334 Data Type: `void`. This command SHALL NOT take any parameters.

2335

2336 **Return Value(s):**

2337 Data Type: `void`. This command SHALL NOT return any value.

2338

2339

2340

2341

2342 **6.2.12 Source.getReadTrigger**

2343 Returns the `Trigger` with the specified name currently associated with this `Source`
2344 object. If no `Trigger` object with the given name exists, the error
2345 `ERROR_TRIGGER_NOT_FOUND` is raised.

2346

2347 **Compliance Requirement:** Compliant systems MAY implement this command.

2348

2349 **Usage:**

2350 `Source.getReadTrigger(name:string) : Trigger`

2351

2352 **Parameter(s):**

2353 name – Data Type: `string`. The name of the `Trigger` object to be returned.

2354

2355 **Return Value(s):**

2356 Data Type: `Trigger`. The requested `Trigger` object.

2357

2358

2359

6.2.13 Source.getAllReadTriggers

Returns all Triggers currently associated with this Source object. If no Triggers are currently associated with this object, the command SHALL complete successfully and an empty list SHALL be returned.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

Source.getAllReadTriggers(void): Trigger[]

Parameter(s):

Data Type: void. This command SHALL NOT take any parameters.

Return Value(s):

Data Type: Trigger[]. The list of all Trigger objects currently associated with this Source.

6.2.14 Source.addTagSelectors

Adds the specified TagSelectors to the list of TagSelectors currently associated with this Source. If one or more of the TagSelectors given are not known, an error SHALL be raised and no TagSelectors SHALL be added. If some of the TagSelectors to be added are already associated with this Source, only the not yet associated TagSelectors shall be added and the command SHALL complete successfully.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

Source.addTagSelectors(selectors:TagSelector[]):void

Parameter(s):

selectors – Data Type: TagSelector[]. The list of TagSelectors to be added.

2395

2396 **Return Value(s):**

2397 Data Type: `void`. This command SHALL NOT return any value.

2398

2399

2400 **6.2.15 Source.removeTagSelectors**

2401 Removes the specified `TagSelectors` from the list of `TagSelectors` currently
2402 associated with this `Source`. If one or more of the `TagSelectors` given are not
2403 known, or if some of the `TagSelectors` to be removed are currently not associated
2404 with this `Source`, these are ignored and all other `TagSelectors` SHALL be removed
2405 and the command SHALL complete successfully.

2406 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
2407 (ERROR_*_NOT_FOUND).

2408

2409 **Compliance Requirement:** Compliant systems MAY implement this command.

2410

2411 **Usage:**

2412 `Source.removeTagSelectors(selectors: TagSelector[]):void`

2413

2414 **Parameter(s):**

2415 `selectors` – Data Type: `TagSelector[]`. The list of `TagSelectors` to be
2416 removed.

2417

2418 **Return Value(s):**

2419 Data Type: `void`. This command SHALL NOT return any value.

2420

2421

2422

2423 **6.2.16 Source.removeAllTagSelectors**

2424 Removes all `TagSelectors` currently associated with this `Source`.

2425 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
2426 (ERROR_*_NOT_FOUND).

2427

2428 **Compliance Requirement:** Compliant systems MAY implement this command.

2429

2430 **Usage:**

2431 `Source.removeAllTagSelectors (void): void`

2432

2433 **Parameter(s):**

2434 Data Type: `void`. This command SHALL NOT take any parameters.

2435

2436 **Return Value(s):**

2437 Data Type: `void`. This command SHALL NOT return any value.

2438

2439

2440

2441 **6.2.17 Source.getTagSelector**

2442 Returns the `TagSelector` with the specified name currently associated with this
2443 `Source` object. If no `TagSelector` object with the given name exists, the error
2444 `ERROR_TAGSELECTOR_NOT_FOUND` is raised.

2445

2446 **Compliance Requirement:** Compliant systems MAY implement this command.

2447

2448 **Usage:**

2449 `Source.getTagSelector(name:string): TagSelector`

2450

2451 **Parameter(s):**

2452 `name` – Data Type: `string`. The name of the `TagSelector` object to be returned.

2453

2454 **Return Value(s):**

2455 Data Type: `TagSelector`. The requested `TagSelector` object.

2456

2457

2458

2459 **6.2.18 Source.getAllTagSelectors**

2460 Returns all TagSelectors currently associated with this Source object. If no
2461 TagSelectors are currently associated with this object, the command SHALL
2462 complete successfully and an empty list SHALL be returned.

2463

2464 **Compliance Requirement:** Compliant systems MAY implement this command.

2465

2466 **Usage:**

2467 Source.getAllTagSelectors(void): TagSelector[]

2468

2469 **Parameter(s):**

2470 Data Type: void. This command SHALL NOT take any parameters.

2471

2472 **Return Value(s):**

2473 Data Type: TagSelector[]. The list of all TagSelector objects currently
2474 associated with this Source.

2475

2476

2477

2478 **6.2.19 Source.getGlimpsedTimeout**

2479 Returns the value of the glimpsedTimeout attribute of this Source object.

2480

2481 **Compliance Requirement:** Compliant systems MAY implement this command.

2482

2483 **Usage:**

2484 Source.getGlimpsedTimeout(void): integer

2485

2486 **Parameter(s):**

2487 Data Type: void. This command SHALL NOT take any parameters.

2488

2489 **Return Value(s):**

2490 Data Type: `integer` . The value of the `glimpsedTimeout` attribute, measured in
2491 milliseconds.

2492

2493

2494

2495

2496 **6.2.20 Source.setGlimpsedTimeout**

2497 Sets the value of the `glimpsedTimeout` attribute of this `Source` object.

2498

2499 **Compliance Requirement:** Compliant systems MAY implement this command.

2500

2501 **Usage:**

2502 `Source.setGlimpsedTimeout(timeout: integer) : void`

2503

2504 **Parameter(s):**

2505 `timeout` - Data Type: `integer` . The value of the `glimpsedTimeout` attribute,
2506 measured in milliseconds. It SHALL be equal to or greater than 0.

2507

2508 **Return Value(s):**

2509 Data Type: `void` . This command SHALL NOT return any value.

2510

2511

2512

2513 **6.2.21 Source.getObservedThreshold**

2514 Returns the value of the `observedThreshold` attribute of this `Source` object.

2515

2516 **Compliance Requirement:** Compliant systems MAY implement this command.

2517

2518 **Usage:**

2519 `Source.getObservedThreshold(void): integer`

2520

2521 **Parameter(s):**

2522 Data Type: `void`. This command SHALL NOT take any parameters.

2523

2524 **Return Value(s):**

2525 Data Type: `integer`. The value of the `observedThreshold` attribute, measured in
2526 milliseconds.

2527

2528

2529

2530

2531 **6.2.22 Source.setObservedThreshold**

2532 Sets the value of the `observedThreshold` attribute of this `Source` object.

2533

2534 **Compliance Requirement:** Compliant systems MAY implement this command.

2535

2536 **Usage:**

2537 `Source.setObservedThreshold(threshold: integer): void`

2538

2539 **Parameter(s):**

2540 `threshold` - Data Type: `integer`. The value of the `observedThreshold`
2541 attribute, measured in milliseconds. It SHALL be equal to or greater than 0.

2542

2543 **Return Value(s):**

2544 Data Type: `void`. This command SHALL NOT return any value.

2545

2546

2547

2548

2549 **6.2.23 Source.getObservedTimeout**

2550 Returns the value of the `observedTimeout` attribute of this `Source` object.

2551

2552 **Compliance Requirement:** Compliant systems MAY implement this command.

2553

2554 **Usage:**
2555 `Source.getObservedTimeout(void): integer`
2556
2557 **Parameter(s):**
2558 Data Type: `void`. This command SHALL NOT take any parameters.
2559
2560 **Return Value(s):**
2561 Data Type: `integer`. The value of the `observedTimeout` attribute, measured in
2562 milliseconds.
2563
2564
2565

2566 **6.2.24 Source.setObservedTimeout**

2567 Sets the value of the `observedTimeout` attribute of this `Source` object.

2568

2569 **Compliance Requirement:** Compliant systems MAY implement this command.

2570

2571 **Usage:**

2572 `Source.setObservedTimeout(timeout: integer): void`

2573

2574 **Parameter(s):**

2575 `timeout` – Data Type: `integer`. The value of the `observedTimeout` attribute,
2576 measured in milliseconds. It SHALL be equal to or greater than 0.

2577

2578 **Return Value(s):**

2579 Data Type: `void`. This command SHALL NOT return any value.

2580

2581

2582

2583 **6.2.25 Source.getLostTimeout**

2584 Returns the value of the `lostTimeout` attribute of this `Source` object.

2585

2586 **Compliance Requirement:** Compliant systems MAY implement this command.

2587

2588 **Usage:**

2589 `Source.getLostTimeout(void): integer`

2590

2591 **Parameter(s):**

2592 Data Type: `void`. This command SHALL NOT take any parameters.

2593

2594 **Return Value(s):**

2595 Data Type: `integer`. The value of the `lostTimeout` attribute, measured in
2596 milliseconds.

2597

2598

2599

2600

2601 **6.2.26 Source.setLostTimeout**

2602 Sets the value of the `lostTimeout` attribute of this `Source` object.

2603

2604 **Compliance Requirement:** Compliant systems MAY implement this command.

2605

2606 **Usage:**

2607 `Source.setLostTimeout(timeout: integer): void`

2608

2609 **Parameter(s):**

2610 `timeout` - Data Type: `integer`. The value of the `lostTimeout` attribute,
2611 measured in milliseconds. It SHALL be equal to or greater than 0.

2612

2613 **Return Value(s):**

2614 Data Type: `void`. This command SHALL NOT return any value.

2615

2616

2617

6.2.27 Source.rawReadIDs

Performs a single read cycle without using the corresponding **Source** object's list of **TagSelectors**. The resulting **ReadReport** SHALL be formatted according to the given **DataSelector**.

Note that this command does not use tag smoothing (see 4.2.1 Smoothing/Event Generation Stage).

Implementations MAY impose restrictions on the usage of this command if **Read Triggers** are active. Such restrictions SHALL be documented.

Executing this command on a **Source** with no **ReadPoint** is an error resulting in **ERROR_READPOINT_NOT_FOUND**.

Compliance Requirement: Compliant systems SHALL implement this command.

Usage:

Source.rawReadIDs(dataSelector: DataSelector): ReadReport

Parameter(s):

dataSelector -Data Type: **DataSelector** . Optional parameter defining the **DataSelector** that should be used to determine what data to report. If no **DataSelector** parameter is given, the Reader's current **DataSelector** SHALL be used.

Return Value(s):

Data Type: **ReadReport** . The resulting **ReadReport**.

6.2.28 Source.readIDs

Performs multiple read cycles using all **TagSelectors** currently associated with this **Source**. The number of read cycles performed SHALL be determined by the value of the **Source** attribute **ReadCyclesPerTrigger**.The resulting **ReadReport** is formatted according to the given **DataSelector**. If a tag is seen in several read cycles, it SHALL only be reported once.

Note that this command does not use event generation.

Implementations MAY impose restrictions on the usage of this command if **Read Triggers** are active. Such restrictions SHALL be documented.

2654 Executing this command on a **Source** with no **ReadPoint** is an error resulting in
 2655 **ERROR_READPOINT_NOT_FOUND**.

2656

2657 **Compliance Requirement:** Compliant systems SHALL implement this command.

2658

2659 **Usage:**

2660 `Source.readIDs(dataSelector: DataSelector): ReadReport`

2661

2662 **Parameter(s):**

2663 `dataSelector` -Data Type: `DataSelector` . Optional parameter defining the
 2664 `DataSelector` that should be used to determine what data to report. If no
 2665 `DataSelector` parameter is given, the Reader's current `DataSelector` SHALL be
 2666 used.

2667

2668 **Return Value(s):**

2669 Data Type: `ReadReport` . The resulting `ReadReport`.

2670

2671

2672

2673 **6.2.29 Source.read**

2674 Performs multiple read cycles on the selected group of tags. The number of read cycles
 2675 performed SHALL be determined by the value of the **Source** attribute
 2676 **ReadCyclesPerTrigger**. The resulting **ReadReport** is formatted according to the
 2677 given **DataSelector**. If a tag is seen in several read cycles, it SHALL only be
 2678 reported once.

2679 Note that this command does not apply tag smoothing (see 4.2.1 Smoothing/Event
 2680 Generation Stage).

2681 An optional list of one or more passwords MAY be specified although the use of read
 2682 passwords is expected to be rare. The application of passwords is dependent upon the
 2683 tag's RF protocol. For instance, for EPCglobal UHF Gen2 tags, the first password
 2684 parameter can be used to access to the tag's kill password that is *read locked*. Such use
 2685 of passwords and their application to specific RF protocols SHALL be documented.

2686 Implementations MAY impose restrictions on the usage of this command if **Read**
 2687 **Triggers** are active. Such restrictions SHALL be documented.

2688 Executing this command on a **Source** with no **ReadPoint** is an error resulting in
 2689 **ERROR_READPOINT_NOT_FOUND**.

2690

2691 **Compliance Requirement:** Compliant systems MAY implement this command.

2692

2693 **Usage:**

```
2694 Source.read(dataSelector: DataSelector,  
2695             passwords: binary[],  
2696             selectors: TagSelector[]): ReadReport
```

2697

2698 **Parameter(s):**

2699 `dataSelector` -Data Type: `DataSelector` . Optional parameter defining the
2700 `DataSelector` that should be used to determine what data to report. If no
2701 `DataSelector` parameter is given, the Reader's current `DataSelector` SHALL be
2702 used.

2703 `passwords` - Data Type: `binary[]` . An optional list of one or more passwords (or
2704 lock code). The use of passwords is dependent upon the tag's RF protocol.

2705 `selectors` - Data Type: `TagSelector` . Optional. If this list is provided then it
2706 MUST be applied as a filter specifying which tags MAY be read.

2707

2708 **Return Value(s):**

2709 Data Type: `ReadReport` . The resulting `ReadReport`.

2710

2711

2712

2713 **6.2.30 Source.writeID**

2714 Programs a tag with the given ID and the optionally specified passwords. A list of
2715 `TagSelector` objects MAY be used to select a single tag, in the reader's field of view,
2716 for writing.

2717 The application of tag filtering (i.e., a list of `TagSelector` objects) is dependent upon
2718 the tag's RF protocol. For instance, EPCglobal UHF Gen2 tags MAY be filtered by their
2719 TAGID memory bank when writing the EPC memory bank. The `Source` object's list of
2720 associated `TagSelectors`, if any, SHALL be ignored during the execution of this
2721 command.

2722 The application of passwords is also dependent upon the tag's RF protocol. For instance,
2723 for EPCglobal Class1 Gen1 tags, the password parameter can correspond to the tag's *lock*
2724 *code* and it can be used to lock a tag for both writing and killing operations. For
2725 EPCglobal UHF Gen2 tags, the first password parameter can correspond to a tag's EPC

2726 *access password*. The second password in the parameter list can correspond to a tag's
2727 *EPC kill password*. Such use of passwords and their application to specific RF protocols
2728 SHALL be documented.

2729 Implementations MAY impose restrictions on the usage of this command if Read
2730 Triggers are active. Such restrictions SHALL be documented.

2731 Executing this command with no tags in the field is an error resulting in
2732 ERROR_NO_TAG.

2733 Executing this command on a Source with no ReadPoint is an error resulting in
2734 ERROR_READPOINT_NOT_FOUND.

2735

2736 **Compliance Requirement:** Compliant systems MAY implement this command.

2737

2738 **Usage:**

```
2739 Source.writeID(ID: binary,  
2740                passwords: binary[],  
2741                selectors: TagSelector[]): void
```

2742

2743 **Parameter(s):**

2744 ID - Data Type: `binary`. The ID to be written to a tag.

2745 passwords - Data Type: `binary[]`. An optional list of one or more passwords (or
2746 lock code). The use of passwords is dependent upon the tag's RF protocol.

2747 selectors - Data Type: `TagSelector`. Optional. If this list is provided then it
2748 MUST be applied as a filter specifying which tags MAY be written.

2749

2750 **Return Value(s):**

2751 Data Type: `void`. This command SHALL NOT return any value.

2752

2753

2754

2755 **6.2.31 Source.write**

2756 Writes the specified TagFieldValues to one or more tags. A list of TagSelector
2757 objects MAY be used to select a set of tags, in the reader's field of view, for writing.

2758 The application of tag filtering (i.e., a list of TagSelector objects) is dependent upon
2759 the tag's RF protocol. For instance, EPCglobal UHF Gen2 tags MAY be filtered by their
2760 TAGID memory bank when writing the EPC memory bank. The Source object's list of

2761 associated **TagSelectors**, if any, SHALL be ignored during the execution of this
 2762 command.

2763 The application of passwords is also dependent upon the tag's RF protocol. For instance,
 2764 for EPCglobal Class1 Gen1 tags, the password parameter can correspond to the tag's *lock*
 2765 *code* and it can be used to lock a tag for both writing and killing operations. For
 2766 EPCglobal UHF Gen2 tags, the first password parameter can correspond to a tag's EPC
 2767 *access password*. The second password in the parameter list can correspond to a tag's
 2768 EPC *kill password*. Such use of passwords and their application to specific RF protocols
 2769 SHALL be documented.

2770 Implementations MAY impose restrictions on the usage of this command if Read
 2771 **Triggers** are active. Such restrictions SHALL be documented.

2772 Executing this command with no tags in the field is an error resulting in
 2773 **ERROR_NO_TAG**.

2774 Executing this command on a **Source** with no **ReadPoint** is an error resulting in
 2775 **ERROR_READPOINT_NOT_FOUND**.

2776

2777 **Compliance Requirement:** Compliant systems MAY implement this command.

2778

2779 **Usage:**

2780 `Source.write(data: TagFieldValue[],`
 2781 `passwords: binary[],`
 2782 `selectors: TagSelector[]): void`

2783

2784 **Parameter(s):**

2785 **data** - Data Type: `TagFieldValue[]`. A list of one or more data field values to be
 2786 written on the selected tags.

2787 **passwords** - Data Type: `binary[]`. An optional list of one or more passwords (or
 2788 lock code). The use of passwords is dependent upon the tag's RF protocol.

2789 **selectors** - Data Type: `TagSelector`. Optional. If this list is provided then it
 2790 MUST be applied as a filter specifying which tags MAY be written.

2791

2792 **Return Value(s):**

2793 Data Type: `void`. This command SHALL NOT return any value.

2794

2795

2796

6.2.32 Source.kill

Kills the specified tag or group of tags. An optional list of `TagSelector` objects can be specified with this command. If specified, the `TagSelector` list parameter overrides the `Source` object's list of `TagSelector` objects that specify the selection of tags to be killed.

A tag's RF protocol may require one or more passwords (or *lock code*) to successfully complete this operation. Such use of passwords and their application to specific RF protocols SHALL be documented.

Implementations MAY impose restrictions on the usage of this command if `Read Triggers` are active. Such restrictions SHALL be documented.

Executing this command with no tags in the field is an error resulting in `ERROR_NO_TAG`.

Executing this command on a `Source` with no `ReadPoint` is an error resulting in `ERROR_READPOINT_NOT_FOUND`.

Compliance Requirement: Compliant systems SHALL implement this command.

Usage:

```
Source.kill(passwords: binary[],  
            selectors: TagSelector[]): void
```

Parameter(s):

`passwords` - Data Type: `binary[]`. Optional. A list of one or more passwords (or *lock code*). The use of passwords is dependent upon the tag's RF protocol. For instance, the EPCglobal UHF Gen2 RF protocol requires both an *access password* and a *kill password* to kill a tag.

`selectors` - Data Type: `TagSelector`. Optional. If this list is provided then it MUST be applied as a filter specifying which tags MAY be killed.

Return Value(s):

Data Type: `void`. This command SHALL NOT return any value.

6.2.33 Source.getReadCyclesPerTrigger

Returns the value of the `readCyclesPerTrigger` attribute of this Source object.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

`Source.getReadCyclesPerTrigger(void): integer`

Parameter(s):

Data Type: `void`. This command SHALL NOT take any parameters.

Return Value(s):

Data Type: `integer`. The value of the `readCyclesPerTrigger` attribute.

6.2.34 Source.setReadCyclesPerTrigger

Sets the value of the `readCyclesPerTrigger` attribute of this Source object.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

`Source.setReadCyclesPerTrigger(cycles : integer):void`

Parameter(s):

`cycles` - Data Type: `integer`. The value of the `readCyclesPerTrigger` attribute. It SHALL be greater than 0.

Return Value(s):

Data Type: `void`. This command SHALL NOT return any value.

2863

2864 **6.2.35 Source.getMaxReadDutyCycle**

2865 Returns the value of the MaxReadDutyCycle attribute of this Source object.

2866

2867 **Compliance Requirement:** Compliant systems MAY implement this command.

2868

2869 **Usage:**

2870 Source.getMaxReadDutyCycle(void): integer

2871

2872 **Parameter(s):**

2873 Data Type: void. This command SHALL NOT take any parameters.

2874

2875 **Return Value(s):**

2876 Data Type: integer. An integer representing the maximum read duty cycle as
2877 percentage is returned. Legal values are between 0 and 100, the default being 100.

2878

2879

2880

2881 **6.2.36 Source.setMaxReadDutyCycle**

2882 Sets the value of the MaxReadDutyCycle attribute of this Source object.

2883

2884 **Compliance Requirement:** Compliant systems MAY implement this command.

2885

2886 **Usage:**

2887 Source.setMaxReadDutyCycle(dutyCycle:integer) : void

2888

2889 **Parameter(s):**

2890 dutyCycle - Data Type: integer. The maximum read duty cycle for this source as
2891 percentage. Legal values are between 0 and 100.

2892

2893 **Return Value(s):**

2894 Data Type: void. This command SHALL NOT return any value.

2895
2896
2897

2898 **6.2.37 Source.getReadTimeout**

2899 Returns the value of the ReadTimeout attribute of this Source object.

2900

2901 **Compliance Requirement:** Compliant systems MAY implement this command.

2902

2903 **Usage:**

2904 Source.getReadTimeout(void): integer

2905

2906 **Parameter(s):**

2907 Data Type: void. This command SHALL NOT take any parameters.

2908

2909 **Return Value(s):**

2910 Data Type: integer . The value of the ReadTimeout attribute, measured in
2911 milliseconds.

2912

2913

2914

2915 **6.2.38 Source.setReadTimeout**

2916 Sets the value of the ReadTimeOut attribute of this Source object.

2917

2918 **Compliance Requirement:** Compliant systems SHALL implement this command.

2919

2920 **Usage:**

2921 Souce.setReadTimeout(timeout: integer): void

2922

2923 **Parameter(s):**

2924 timeout - Data Type: integer . The value of the ReadTimeout attribute,
2925 measured in milliseconds. It SHALL be equal to or greater than 0.

2926

2927 **Return Value(s):**

2928 Data Type: `void`. This command SHALL NOT return any value.

2929

2930

2931

2932 **6.2.39 Source.getSession**

2933 Returns the value of the corresponding `Source` object's `Session` attribute. The
2934 application of this attribute is dependent upon a tag's RF protocol. The use of this
2935 attribute and its application to specific RF protocols SHALL be documented.

2936

2937 **Compliance Requirement:** Compliant systems MAY implement this command.

2938

2939 **Usage:**

2940 `Source.getSession(void): integer`

2941

2942 **Parameter(s):**

2943 Data Type: `void`. This command SHALL NOT take any parameters.

2944

2945 **Return Value(s):**

2946 Data Type: `integer`. The value of the `Session` attribute.

2947

2948

2949

2950 **6.2.40 Source.setSession**

2951 Sets the value of the corresponding `Source` object's `Session` attribute. The application
2952 of this attribute is dependent upon a tag's RF protocol. The use of this attribute and its
2953 application to specific RF protocols SHALL be documented.

2954

2955 **Compliance Requirement:** Compliant systems MAY implement this command.

2956

2957 **Usage:**

2958 `Souce.setSession(session: integer): void`

2959

2960 **Parameter(s):**
2961 `session` - Data Type: `integer` . The value of the `Session` attribute associated with
2962 the corresponding `Source` object.

2963
2964 **Return Value(s):**
2965 Data Type: `void` . This command SHALL NOT return any value.

2966
2967
2968

2969 **6.2.41 Source.getMaxNumberSupported**

2970 Returns the maximum number of `Sources` this Reader supports. This is a static
2971 method.

2972

2973 **Compliance Requirement:** Compliant systems SHALL implement this command.

2974

2975 **Usage:**

2976 `static Source.getMaxNumberSupported ( ) : integer`

2977

2978 **Parameter(s):**

2979 Data Type: `void` . This command SHALL NOT take any parameters.

2980

2981 **Return Value(s):**

2982 Data Type: `integer` . The maximum number of `Sources` the Reader supports, or `-1` if
2983 unbounded (no limit).

2984

2985

2986

2987 **6.3 Object TagFieldValue**

2988 This object is not created on the reader. It is only passed as a parameter to the `Source`
2989 command `write`. This object encapsulates the information for a named tag data field
2990 and its associated data value.

2991

2992 **6.3.1 TagFieldValue.TagFieldName:String**

2993 The name of a TagField object that must already exist on the reader. The TagField
2994 object SHALL be associated with the corresponding Source object. The TagField
2995 object specifies the tag memory location for the corresponding data field.

2997 **6.3.2 TagFieldValue.Value:binary**

2998 The data field's corresponding data value.

3001 **6.4 Object ReadPoint**

3003 **6.4.1 ReadPoint.getName**

3004 Returns the name of the given ReadPoint object.

3006 **Compliance Requirement:** Compliant systems SHALL implement this command.

3008 **Usage:**

3009 `ReadPoint.getName(void): string`

3011 **Parameter(s):**

3012 Data Type: `void`. This command SHALL NOT take any parameters.

3013 **Return Value(s):**

3014 Data Type: `string`. The name of the ReadPoint object is returned.

3016 **6.5 Object Trigger**

3017 Trigger is a non-mutable object that can exist as a composite member of either a Source
3018 or a NotificationChannel object, i.e., Trigger objects are added and removed from
3019 Source and NotificationChannel objects.

3020 There is one pool of Trigger objects. Source and NotificationChannel
3021 objects MAY both reference the same Trigger object. Source and
3022 NotificationChannel objects MAY contain zero or up to a predefined maximum
3023 number of total Trigger member objects. Source and NotificationChannel
3024 objects SHALL contain zero Trigger objects by default.

3025 A Trigger SHALL be activated as soon as it is added to another object. It SHALL be
 3026 deactivated once it has been removed from all objects it had been added to, i.e. when it is
 3027 not associated with any other object.

3028

3029 The following table shows the legal values for `triggervalue` and associated
 3030 `TriggerType`:

triggervalue	TriggerType	Description
n/a	continuous	When associated with a Source, the trigger fires repeatedly, as rapidly as possible. ² When associated with a NotificationChannel the trigger fires whenever the the report queue transitions from empty to non-empty.
ms=<n>	timer	Periodic interval trigger; <n> is the number of milliseconds for the interval.
type=<rising falling>;port=<n>;pin=<n>	ioEdge	The trigger fires when the specified i/o pin on the specified port changes state. The type SHALL be 'rising' or 'falling'.
port=<n>;value=<hex> [;mask=<hex>]	ioValue	The trigger fires when the specified port input data has a specific value. If a mask is specified than it is bitwise "anded" with the port input data before comparing to the value. The value and mask are specified as hexadecimal (e.g.

² As rapidly as possible, accounting for read duty cycles and other configuration settings as appropriate.

		“...;value=00F2”).
vendor=<name>; type=<type> [;...arbitrary key/value pairs...]	vendorExtension	Non-standard vendor extensions. The <name> is required and SHALL contain the vendor’s name. The <type> indicates the type of trigger; the actual value is vendor-dependent. Additional key/value pairs MAY be specified, and are vendor-dependent.

Table 7: Possible Trigger values

6.5.1 Trigger.create

Create a **Trigger** object with a given name. If a **Trigger** object with the same name exists already, an error is returned. This is a static method.

The **Trigger** object’s type and value establish the conditions whereby the trigger SHALL fire. See the table above for a description of the different possible trigger conditions.

If the Reader doesn’t support the requested **TriggerType**, the error condition **ERROR_PARAMETER_NOT_SUPPORTED** shall be raised.

The **Trigger** SHALL implicitly be added to the list of all **Triggers** kept by the **ReaderDevice** object.

The newly created object has its standard properties (if supported) initialized as:

Property	Value
Name	name
Type	type
Value	value

Table 8: Default properties for Trigger object

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
static Trigger.create (
    name: string,
```

3050 type: TriggerType,
3051 value: triggervalue): Trigger

3052

3053 **Parameter(s):**

3054 name – Data Type: string . The name of the object to be created.

3055 type – Data Type: TriggerType. An enumerated value representing the type of
3056 trigger specified (see table above).

3057 value – Data Type: triggervalue. A string that's formatted according to the
3058 syntax specified in the table above. This parameter represents the trigger characteristics
3059 as they apply to the trigger type specified.

3060

3061 **Return Value(s):**

3062 Data Type: Trigger . The newly created object.

3063

3064

3065

3066 **6.5.2 Trigger.getMaxNumberSupported**

3067 Returns the maximum number of Triggers this Reader supports. This is a static
3068 method.

3069

3070 **Compliance Requirement:** Compliant systems SHALL implement this command.

3071

3072 **Usage:**

3073 static Trigger.getMaxNumberSupported(): integer

3074

3075 **Parameter(s):**

3076 Data Type: void . This command SHALL NOT take any parameters.

3077

3078 **Return Value(s):**

3079 Data Type: integer . The maximum number of Triggers the Reader supports, or -1 if
3080 unbounded (no limit).

3081

3082

3083

3084 **6.5.3 Trigger.getName**

3085 Returns the name of the given `Trigger` object.

3086

3087 **Compliance Requirement:** Compliant systems MAY implement this command.

3088

3089 **Usage:**

3090 `Trigger.getName()`: `string`

3091

3092 **Parameter(s):**

3093 Data Type: `void`. This command SHALL NOT take any parameters.

3094

3095 **Return Value(s):**

3096 Data Type: `string`. The name of the `Trigger` object is returned.

3097

3098

3099

3100 **6.5.4 Trigger.getType**

3101 Returns the `TriggerType` of this `Trigger`.

3102

3103 **Compliance Requirement:** Compliant systems MAY implement this command.

3104

3105 **Usage:**

3106 `Trigger.getType()`: `TriggerType`

3107

3108 **Parameter(s):**

3109 Data Type: `void`. This command SHALL NOT take any parameters.

3110

3111 **Return Value(s):**

3112 Data Type: `TriggerType`. `TriggerType` of this `Trigger`.

3113

3114

3115

6.5.5 Trigger.getValue

Returns the `triggervalue` of this `Trigger` object.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

`Trigger.getValue ()`: `triggervalue`

Parameter(s):

Data Type: `void`. This command SHALL NOT take any parameters.

Return Value(s):

Data Type: `triggervalue`. The `triggervalue` of this `Trigger`.

6.5.6 Trigger.fire

This command causes this `Trigger` object to *fire* (i.e., activate the trigger) immediately, regardless of the value settings of the `Trigger`.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

`Trigger.fire ()`: `void`

Parameter(s):

Data Type: `void`. This command SHALL NOT take any parameters.

Return Value(s):

Data Type: `void`. This command SHALL NOT return any value.

3148

3149 **6.6 Object TagSelector**

3150 A TagSelector is a non-mutable object that MAY exist as a member of a Source
3151 object (i.e., TagSelector objects are added and removed from a Source object). A
3152 Source object can contain zero or more TagSelector objects. A Source object
3153 SHALL contain zero TagSelector objects by default.

3154 The definition of how TagSelectors filter tags is described in section 5.4
3155 TagSelectors.

3156 **6.6.1 TagSelector.create**

3157 Create a TagSelector object with a given name. If a TagSelector object with the
3158 same name exists already, an error is returned. This is a static method.

3159 The TagSelector SHALL implicitly be added to the list of all TagSelectors kept
3160 by the ReaderDevice object.

3161 The newly created object has its standard properties (if supported) initialized as:

Property	Value
Name	name
TagField	field
Value	value
Mask	mask
InclusiveFlag	inclusiveFlag

3162 **Table 9: Default properties for TagSelector object**

3163

3164 **Compliance Requirement:** Compliant systems MAY implement this command.

3165

3166 **Usage:**

```
3167 static TagSelector.create (  
3168     name: string,  
3169     field: TagField,  
3170     value: hexstring,  
3171     mask: hexstring,  
3172     inclusiveFlag: boolean): TagSelector
```

3173

3174 **Parameter(s):**

3175 **name** – Data Type: `string`. The name of the object to be created.
3176 **field** – Data Type: `TagField`. The tag field to be used by this tag selector.
3177 **value** – Data Type: `hexstring`. A hexadecimal value representing the tag ID *value*
3178 that is used to filter tag data (see read filter specification in section 5.4).
3179 **mask** – Data Type: `hexstring`. A hexadecimal mask representing a tag ID *mask* that
3180 is used to filter tag data (see tag read filter specification in section 5.4).
3181 **inclusiveFlag** – Data Type: `boolean`. A *flag* indicating whether the filter is
3182 inclusive or exclusive (see read filter specification in section 5.4).

3183

3184 **Return Value(s):**

3185 Data Type: `TagSelector`. The created `TagSelector` object is returned.

3186

3187

3188

3189 **6.6.2 TagSelector.getMaxNumberSupported**

3190 Returns the maximum number of `TagSelector`s this Reader supports. This is a static
3191 method.

3192

3193 **Compliance Requirement:** Compliant systems SHALL implement this command.

3194

3195 **Usage:**

3196 `TagSelector.getMaxNumberSupported()`: `integer`

3197

3198 **Parameter(s):**

3199 Data Type: `void`. This command SHALL NOT take any parameters.

3200

3201 **Return Value(s):**

3202 Data Type: `integer`. The maximum number of `TagSelector`s the Reader supports,
3203 or `-1` if unbounded (no limit).

3204

3205

3206 **6.6.3 TagSelector.getName**

3207 Returns the name of the given `TagSelector` object.

3208

3209 **Compliance Requirement:** Compliant systems MAY implement this command.

3210

3211 **Usage:**

3212 `TagSelector.getName(): string`

3213

3214 **Parameter(s):**

3215 Data Type: `void`. This command SHALL NOT take any parameters.

3216

3217 **Return Value(s):**

3218 Data Type: `string`. The name of the `TagSelector` object is returned.

3219

3220

3221

3222 **6.6.4 TagSelector.getTagField**

3223 Returns the tag field attribute of this `TagSelector` object.

3224

3225 **Compliance Requirement:** Compliant systems MAY implement this command.

3226

3227 **Usage:**

3228 `TagSelector.getTagField(): TagField`

3229

3230 **Parameter(s):**

3231 Data Type: `void`. This command SHALL NOT take any parameters.

3232

3233 **Return Value(s):**

3234 Data Type: `TagField`. The tag field associated with this `TagSelector`.

3235

3236

3237

3238 **6.6.5 TagSelector.getValue**

3239 Returns the value attribute of this `TagSelector` object.

3240

3241 **Compliance Requirement:** Compliant systems MAY implement this command.

3242

3243 **Usage:**

3244 `TagSelector.getValue(): hexstring`

3245

3246 **Parameter(s):**

3247 Data Type: `void`. This command SHALL NOT take any parameters.

3248

3249 **Return Value(s):**

3250 Data Type: `hexstring`. The `TagSelector` value attribute.

3251

3252

3253 **6.6.6 TagSelector.getMask**

3254 Returns the mask attribute of this `TagSelector` object.

3255

3256 **Compliance Requirement:** Compliant systems MAY implement this command.

3257

3258 **Usage:**

3259 `TagSelector.getMask(): hexstring`

3260

3261 **Parameter(s):**

3262 Data Type: `void`. This command SHALL NOT take any parameters.

3263

3264 **Return Value(s):**

3265 Data Type: `hexstring`. The `TagSelector` mask attribute.

3266

3267

3268 **6.6.7 TagSelector.getInclusiveFlag**

3269 Returns the inclusiveFlag attribute of this `TagSelector` object.

3270

3271 **Compliance Requirement:** Compliant systems MAY implement this command.

3272

3273 **Usage:**

3274 `TagSelector.getInclusiveFlag(): boolean`

3275

3276 **Parameter(s):**

3277 Data Type: `void`. This command SHALL NOT take any parameters.

3278

3279 **Return Value(s):**

3280 Data Type: `boolean`. The “true”, if the `TagSelector` is inclusive, “false”
3281 otherwise.

3282

3283

3284

3285 **6.7 Object CommandChannel**

3286 There are no explicit operations on this object. A compliant Reader SHALL however
3287 have at least one command channel with a default address where it is listening to
3288 incoming commands. This default address SHALL be listed in the documentation of a
3289 compliant reader.

3290

3291 **6.8 Object NotificationChannel**

3292 A `NotificationChannel` is used to report notifications to a single host. It has a list
3293 of `Sources` associated with it. Only events coming from these `Sources` SHALL be
3294 reported. It also has a `DataSelector` that determines what data should be reported to the
3295 host. If no `DataSelector` is explicitly associated, then the Reader’s *current*
3296 `DataSelector` SHALL be used.

3297 Notification SHALL be sent whenever the associated `Trigger` fires. Alternatively, a
3298 host MAY also query the `NotificationChannel` for its contents, using the
3299 command `NotificationChannel.readQueuedData()`.

3300

3301 **6.8.1 NotificationChannel.create**

3302 Create a `NotificationChannel` object with a given name. If a
3303 `NotificationChannel` object with the same name exists already, an error is
3304 returned. This is a static method.

3305 The `NotificationChannel` SHALL implicitly be added to the list of all
3306 `NotificationChannels` kept by the `ReaderDevice` object.

3307 The newly created object has its standard properties (if supported) initialized as:

Property	Value
Name	name
Address	addr

3308 **Table 10: Default properties for NotificationChannel object**

3309

3310 **Compliance Requirement:** Compliant systems MAY implement this command.

3311

3312 **Usage:**

```
3313 static NotificationChannel.create(  
3314     name: String,  
3315     addr: address): NotificationChannel
```

3316

3317 **Parameter(s):**

3318 name - Data Type: `string`. The name of the `NotificationChannel` to be created.

3319 addr - Data Type: `address`. The address of where notifications will be sent to. See
3320 section 9.1 *Address Notation* for more details about addresses. If the address *mode*
3321 option is omitted, then mode *connect* is implied. If mode *listen* is specified, then the host
3322 should execute the `getEffectiveAddress` command to determine the port assigned
3323 by the Reader for listening for a connection from the host.

3324

3325 **Return Value(s):**

3326 Data Type: `NotificationChannel`. The newly created object.

3327

3328

3329 **6.8.2 NotificationChannel.getName**

3330 Returns the name of the given `NotificationChannel` object.

3331

3332 **Compliance Requirement:** Compliant systems MAY implement this command.

3333

3334 **Usage:**

```
3335 NotificationChannel.getName(void): string
```

3336

3337 **Parameter(s):**

3338 Data Type: `void`. This command SHALL NOT take any parameters.

3339

3340 **Return Value(s):**

3341 Data Type: `string`. The name of the `NotificationChannel` object is returned.

3342

3343

3344 **6.8.3 NotificationChannel.getAddress**

3345 Returns the address to which this `NotificationChannel` object sends its
3346 notifications as specified by either commands `NotificationChannel.create` or
3347 `NotificationChannel.setAddress`. See also `getEffectiveAddress`.

3348

3349 **Compliance Requirement:** Compliant systems MAY implement this command.

3350

3351 **Usage:**

3352 `NotificationChannel.getAddress(void): address`

3353

3354 **Parameter(s):**

3355 Data Type: `void`. This command SHALL NOT take any parameters.

3356

3357 **Return Value(s):**

3358 Data Type: `address`. The reporting address for this `NotificationChannel` as
3359 specified by either commands `NotificationChannel.create` or
3360 `NotificationChannel.setAddress`. See section 9.1 *Address Notation* for
3361 more details about addresses.

3362

3363

3364 **6.8.4 NotificationChannel.getEffectiveAddress**

3365 Returns the effective address to which this `NotificationChannel` object sends its
3366 notifications.

3367

3368 **Compliance Requirement:** Compliant systems MAY implement this command.

3369

3370 **Usage:**
3371 `NotificationChannel.getEffectiveAddress(void): address`
3372
3373 **Parameter(s):**
3374 Data Type: `void`. This command SHALL NOT take any parameters.
3375
3376 **Return Value(s):**
3377 Data Type: `address`. The effective address for this `NotificationChannel`.
3378 Example: "`tcp://*:9090?mode=listen`". In this example, the channel was set to
3379 listen for a TCP connection. The Reader assigned port 9090 to this request and therefore
3380 port 9090 is returned as the effective address. See section 9.1 *Address Notation* for more
3381 details about addresses.
3382
3383

3384 **6.8.5 NotificationChannel.setAddress**

3385 Sets the address to which this `NotificationChannel` object sends its notifications.

3386

3387 **Compliance Requirement:** Compliant systems MAY implement this command.

3388

3389 **Usage:**

3390 `NotificationChannel.setAddress(addr: address) : integer`

3391

3392 **Parameter(s):**

3393 `addr` - Data Type: `address`. The address of where notifications will be sent to. See
3394 section 9.1 *Address Notation* for more details about addresses. If the address *mode*
3395 option is omitted, then mode *connect* is implied. If mode *listen* is specified, then the host
3396 should execute the `getEffectiveAddress` command to determine the port assigned
3397 by the Reader for listening for a connection from the host.

3398

3399 **Return Value(s):**

3400 Data Type: `integer`. If *connect mode* was specified, then the return value does not
3401 apply and the Reader SHALL return zero. Otherwise, if *listen mode* was specified, then
3402 the return SHALL return the *port number* assigned by the Reader to listen for host
3403 `Notification Channel` connections.

3404

3405

3406

3407 **6.8.6 NotificationChannel.getDataSelector**

3408 Returns the `DataSelector` currently associated with this `NotificationChannel`
3409 object. If there is no `DataSelector` object associated, the error
3410 `ERROR_DATASELECTOR_NOT_FOUND` is raised.

3411

3412 **Compliance Requirement:** Compliant systems MAY implement this command.

3413

3414 **Usage:**

3415 `NotificationChannel.getDataSelector(void): DataSelector`

3416

3417 **Parameter(s):**

3418 Data Type: `void`. This command SHALL NOT take any parameters.

3419

3420 **Return Value(s):**

3421 Data Type: `DataSelector`. The currently associated `DataSelector` object.

3422

3423

3424

3425 **6.8.7 NotificationChannel.setDataSelector**

3426 Sets the `DataSelector` object to be used with this `NotificationChannel`.

3427

3428 **Compliance Requirement:** Compliant systems MAY implement this command.

3429

3430 **Usage:**

3431 `NotificationChannel.setDataSelector (`
3432 `dataSelector: DataSelector): void`

3433

3434 **Parameter(s):**

3435 `dataSelector` – Data Type: `DataSelector`. The `DataSelector` to be
3436 assigned to this `NotificationChannel`.

3437

3438 **Return Value(s):**

3439 Data Type: `void`. This command SHALL NOT return any value.

3440

3441

3442

3443 **6.8.8 NotificationChannel.addSources**

3444 Adds the specified `Sources` to the list of `Sources` currently associated with this
3445 `NotificationChannel`. If one or more of the `Sources` given are not known, an
3446 error SHALL be raised and no `Sources` SHALL be added. If some of the `Sources` to
3447 be added are already associated with this `NotificationChannel`, only the not yet
3448 associated `Sources` shall be added and the command SHALL complete successfully.

3449

3450 **Compliance Requirement:** Compliant systems MAY implement this command.

3451

3452 **Usage:**

3453 `NotificationChannel.addSources(sources: Source[]): void`

3454

3455 **Parameter(s):**

3456 `sources`- Data Type: `Source[]`. The list of `Sources` to be added.

3457

3458 **Return Value(s):**

3459 Data Type: `void`. This command SHALL NOT return any value.

3460

3461

3462

3463 **6.8.9 NotificationChannel.removeSources**

3464 Removes the specified `Sources` from the list of `Sources` currently associated with
3465 this `NotificationChannel`. If one or more of the `Sources` given are not known,
3466 or if some of the `Sources` to be removed are currently not associated with this
3467 `NotificationChannel`, these are ignored and all other `Sources` SHALL be
3468 removed and the command SHALL complete successfully.

3469 Removing a 'default' object changes the 'default' to `null`. Using a '`null`' object is an error
3470 (`ERROR_*_NOT_FOUND`).

3471

3472 **Compliance Requirement:** Compliant systems MAY implement this command.
3473
3474 **Usage:**
3475 `NotificationChannel.removeSources(sources: Source[]): void`
3476
3477 **Parameter(s):**
3478 `sources` – Data Type: `Source[]`. The list of `Sources` to be removed.
3479
3480 **Return Value(s):**
3481 Data Type: `void`. This command SHALL NOT return any value.
3482
3483
3484

3485 **6.8.10 NotificationChannel.removeAllSources**
3486 Removes all `Sources` currently associated with this `NotificationChannel`.
3487 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
3488 (`ERROR_*_NOT_FOUND`).
3489

3490 **Compliance Requirement:** Compliant systems MAY implement this command.

3491
3492 **Usage:**
3493 `NotificationChannel.removeAllSources(void): void`
3494

3495 **Parameter(s):**
3496 Data Type: `void`. This command SHALL NOT take any parameters.
3497

3498 **Return Value(s):**
3499 Data Type: `void`. This command SHALL NOT return any value.

3500
3501
3502

6.8.11 NotificationChannel.getSource

Returns the `Source` with the specified name currently associated with this `NotificationChannel` object. If no `Source` object with the given name exists, the error `ERROR_SOURCE_NOT_FOUND` is raised.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

`NotificationChannel.getSource(name: string): Source`

Parameter(s):

`name` – Data Type: `string`. The name of the `Source` object to be returned.

Return Value(s):

Data Type: `Source`. The requested `Source` object.

6.8.12 NotificationChannel.getAllSources

Returns all `Sources` currently associated with this `NotificationChannel` object. If no `Sources` are currently associated with this object, the command SHALL complete successfully and an empty list SHALL be returned.

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

`NotificationChannel.getAllSources(void): Source[]`

Parameter(s):

Data Type: `void`. This command SHALL NOT take any parameters.

Return Value(s):

3535 Data Type: `Source[]`. The list of all `Source` objects currently associated with this
3536 `NotificationChannel`.

3537

3538

3539

3540 **6.8.13 NotificationChannel.addNotificationTriggers**

3541 Adds the specified `Triggers` to the list of `Notification Triggers` currently associated
3542 with this `NotificationChannel`. If one or more of the `Triggers` given are not
3543 supported or are not known, an error SHALL be raised and no `Triggers` SHALL be
3544 added. If some of the `Triggers` to be added are already associated with this
3545 `NotificationChannel`, only the not yet associated `Triggers` shall be added and
3546 the command SHALL complete successfully.

3547 Once a `Trigger` has been added, it is immediately activated.

3548

3549 **Compliance Requirement:** Compliant systems MAY implement this command.

3550

3551 **Usage:**

3552 `NotificationChannel.addNotificationTriggers(
3553 triggers: Trigger[]): void`

3554

3555 **Parameter(s):**

3556 `triggers` – Data Type: `Trigger[]`. The list of `Triggers` to be added.

3557

3558 **Return Value(s):**

3559 Data Type: `void`. This command SHALL NOT return any value.

3560

3561

3562

3563 **6.8.14 NotificationChannel.removeNotificationTriggers**

3564 Removes the specified `Triggers` from the list of `Notification Triggers` currently
3565 associated with this `NotificationChannel`. If one or more of the `Triggers` given
3566 are not supported or not known, or if some of the `Triggers` to be removed are currently
3567 not associated with this `NotificationChannel`, these are ignored and all other
3568 `Triggers` SHALL be removed and the command SHALL complete successfully.

3569 Once a `Trigger` isn't associated to any object anymore, it is deactivated.

3570 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
3571 (ERROR_*_NOT_FOUND).

3572

3573 **Compliance Requirement:** Compliant systems MAY implement this command.

3574

3575 **Usage:**

3576 `NotificationChannel.removeNotificationTriggers(
3577 triggers: Trigger[]): void`

3578

3579 **Parameter(s):**

3580 `triggers` – Data Type: `Trigger[]`. The list of `Triggers` to be removed.

3581

3582 **Return Value(s):**

3583 Data Type: `void`. This command SHALL NOT return any value.

3584

3585

3586

3587 **6.8.15 NotificationChannel.removeAllNotificationTriggers**

3588 Removes all `Triggers` currently associated with this `NotificationChannel`.

3589 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
3590 (ERROR_*_NOT_FOUND).

3591

3592 **Compliance Requirement:** Compliant systems MAY implement this command.

3593

3594 **Usage:**

3595 `NotificationChannel.removeAllNotificationTriggers(
3596 void): void`

3597

3598 **Parameter(s):**

3599 Data Type: `void`. This command SHALL NOT take any parameters.

3600

3601 **Return Value(s):**

3602 Data Type: `void`. This command SHALL NOT return any value.

3603

3604

3605

3606 **6.8.16 NotificationChannel.getNotificationTrigger**

3607 Returns the `Trigger` with the specified name currently associated with this
3608 `NotificationChannel` object. If no `Trigger` object with the given name exists,
3609 the error `ERROR_TRIGGER_NOT_FOUND` is raised.

3610

3611 **Compliance Requirement:** Compliant systems MAY implement this command.

3612

3613 **Usage:**

3614 `NotificationChannel.getNotificationTrigger(
3615 name: string): Trigger`

3616

3617 **Parameter(s):**

3618 name – Data Type: `string`. The name of the `Trigger` object to be returned.

3619

3620 **Return Value(s):**

3621 Data Type: `Trigger`. The requested `Trigger` object.

3622

3623

3624

3625 **6.8.17 NotificationChannel.getAllNotificationTriggers**

3626 Returns all `Triggers` currently associated with this `NotificationChannel` object.
3627 If no `Triggers` are currently associated with this object, the command SHALL
3628 complete successfully and an empty list SHALL be returned.

3629

3630 **Compliance Requirement:** Compliant systems MAY implement this command.

3631

3632 **Usage:**

3633 `NotificationChannel.getAllNotificationTriggers(
3634 void): Trigger[]`

3635

3636 **Parameter(s):**

3637 Data Type: `void`. This command SHALL NOT take any parameters.

3638

3639 **Return Value(s):**

3640 Data Type: `Trigger[]`. The list of all `Trigger` objects currently associated with this
3641 `NotificationChannel`.

3642

3643

3644

3645 **6.8.18 NotificationChannel.readQueuedData**

3646 This command returns the data currently queued for delivery in the report buffer. What
3647 data is reported depends on the `DataSelector` associated with the
3648 `NotificationChannel`.

3649 Implementations MAY impose restrictions on the usage of this command if `Notification`
3650 `Triggers` are active. Such restrictions SHALL be documented.

3651

3652 **Compliance Requirement:** Compliant systems MAY implement this command.

3653

3654 **Usage:**

3655 `NotificationChannel.readQueuedData(`
3656 `clearBuffer: boolean): ReadReport`

3657

3658 **Parameter(s):**

3659 `clearBuffer` -Data Type: `boolean`. An optional flag to indicate if the report buffer
3660 should be cleared after the `ReadReport` is returned. The default SHALL be `true`.

3661

3662 **Return Value(s):**

3663 Data Type: `ReadReport`. The resulting `ReadReport`.

3664

3665

3666

3667 **6.8.19 NotificationChannel.getMaxNumberSupported**

3668 Returns the maximum number of `NotificationChannels` this Reader supports.
3669 This is a static method.

3670

3671 **Compliance Requirement:** Compliant systems SHALL implement this command.

3672

3673 **Usage:**

3674 `static NotificationChannel.getMaxNumberSupported(): integer`

3675

3676 **Parameter(s):**

3677 Data Type: `void`. This command SHALL NOT take any parameters.

3678

3679 **Return Value(s):**

3680 Data Type: `integer`. The maximum number of Notification Channels the Reader
3681 supports, or -1 if unbounded (no limit).

3682

3683

3684

3685 **6.9 Object DataSelector**

3686 DataSelector objects are used to define what data SHALL be reported in notification
3687 messages or by commands that take a DataSelector object as parameter. Only events
3688 and data fields that are added to the respective lists SHALL be reported.

3689

3690 **6.9.1 DataSelector.create**

3691 Create a DataSelector object with a given name. If a DataSelector object with
3692 the same name exists already, an error is returned. This is a static method.

3693 The DataSelector SHALL implicitly be added to the list of all DataSelectors
3694 kept by the ReaderDevice object.

3695 The newly created object has its standard properties initialized as:

Property	Value
Name	name
Fieldnames	[]
EventFilters	[]
TagFieldNames	[]

3696 **Table 11: Default properties for DataSelector object**

3697

3698 **Compliance Requirement:** Compliant systems MAY implement this command.

3699

3700 **Usage:**

3701 `static DataSelector.create (name: string): DataSelector`

3702

3703 **Parameter(s):**

3704 `name` – Data Type: `string`. The name of the object to be created.

3705

3706 **Return Value(s):**

3707 Data Type: `DataSelector`. The newly created object.

3708

3709

3710 **6.9.2 DataSelector.getName**

3711 Returns the name of the given `DataSelector` object.

3712

3713 **Compliance Requirement:** Compliant systems MAY implement this command.

3714

3715 **Usage:**

3716 `DataSelector.getName (void): string`

3717

3718 **Parameter(s):**

3719 Data Type: `void`. This command SHALL NOT take any parameters.

3720

3721 **Return Value(s):**

3722 Data Type: `string`. The name of the `DataSelector` object is returned.

3723

3724

3725 **6.9.3 DataSelector.addFieldNames**

3726 Adds the specified field names to the list of field names currently associated with this
3727 `DataSelector`. If one or more of the `FieldNames` given are not supported, an error
3728 SHALL be raised and no `FieldNames` SHALL be added. If some of the `FieldNames`
3729 to be added are already associated with this `DataSelector`, only the not yet associated
3730 `FieldNames` shall be added and the command SHALL complete successfully.

3731

3732 **Compliance Requirement:** Compliant systems MAY implement this command.

3733

3734 **Usage:**

3735 `DataSelector.addFieldNames (fields: FieldName[]): void`

3736

3737 **Parameter(s):**

3738 `fields` – Data Type: `FieldName[]`. The list of `FieldNames` to be added. These
3739 data fields SHALL from now on be reported.

3740

3741 **Return Value(s):**

3742 Data Type: `void`. This command SHALL NOT return any value.

3743

3744

3745 **6.9.4 DataSelector.removeFieldNames**

3746 Removes the specified field names from the list of field names currently associated with
3747 this `DataSelector`. If one or more of the `FieldNames` given are not supported,
3748 these are ignored and all other `FieldNames` SHALL be removed and the command
3749 SHALL complete successfully.

3750 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
3751 (`ERROR_*_NOT_FOUND`).

3752

3753 **Compliance Requirement:** Compliant systems MAY implement this command.

3754

3755 **Usage:**

3756 `DataSelector.removeFieldNames (fields: FieldName[]): void`

3757

3758 **Parameter(s):**

3759 `fields` – Data Type: `FieldName[]`. The list of `FieldNames` to be removed.
3760 These data fields SHALL NOT be reported anymore.

3761

3762 **Return Value(s):**

3763 Data Type: `void`. This command SHALL NOT return any value.

3764

3765

3766 **6.9.5 DataSelector.removeAllFieldNames**

3767 Removes all field names currently associated with this `DataSelector`.

3768 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
3769 (ERROR_*_NOT_FOUND).

3770

3771 **Compliance Requirement:** Compliant systems MAY implement this command.

3772

3773 **Usage:**

3774 `DataSelector.removeAllFieldNames(void): void`

3775

3776 **Parameter(s):**

3777 Data Type: `void`. This command SHALL NOT take any parameters.

3778

3779 **Return Value(s):**

3780 Data Type: `void`. This command SHALL NOT return any value.

3781

3782

3783 **6.9.6 DataSelector.getAllFieldNames**

3784 Returns all field names currently associated with this `DataSelector`.

3785

3786 **Compliance Requirement:** Compliant systems MAY implement this command.

3787

3788 **Usage:**

3789 `DataSelector.getAllFieldNames(void): fieldName[]`

3790

3791 **Parameter(s):**

3792 Data Type: `void`. This command SHALL NOT take any parameters.

3793

3794 **Return Value(s):**

3795 Data Type: `fieldName[]`. The list of all `fieldName`s currently being reported.

3796

3797

3798 **6.9.7 DataSelector.addEventFilters**

3799 Adds the specified **EventTypes** to the list of **EventTypes** currently associated with
3800 this **DataSelector**. If one or more of the **EventTypes** given are not supported, an
3801 error SHALL be raised and no **EventTypes** SHALL be added. If some of the
3802 **EventTypes** to be added are already associated with this **DataSelector**, only the
3803 not yet associated **EventTypes** shall be added and the command SHALL complete
3804 successfully.

3805 Only events of the types that have been added SHALL be reported. That is, the
3806 **EventTypes** added are to be used as *inclusive* filters.

3807

3808 **Compliance Requirement:** Compliant systems MAY implement this command.

3809

3810 **Usage:**

3811 `DataSelector.addEventFilters (events: EventType[]): void`

3812

3813 **Parameter(s):**

3814 **events** – Data Type: `EventType[]`. The list of **EventTypes** to be added. Events
3815 of these types SHALL from now on be reported.

3816

3817 **Return Value(s):**

3818 Data Type: `void`. This command SHALL NOT return a value.

3819

3820

3821 **6.9.8 DataSelector.removeEventFilters**

3822 Removes the specified **EventTypes** from the list of (inclusive) event filters currently
3823 associated with this **DataSelector**. If one or more of the **EventTypes** given are not
3824 supported, or if some of the **EventTypes** to be removed are currently not associated
3825 with this **DataSelector**, these are ignored and all other **EventTypes** SHALL be
3826 removed and the command SHALL complete successfully.

3827 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
3828 (`ERROR_*_NOT_FOUND`).

3829

3830 **Compliance Requirement:** Compliant systems MAY implement this command.

3831

3832 **Usage:**
3833 `DataSelector.removeEventFilters (events: EventType[]): void`

3834

3835 **Parameter(s):**

3836 `events` – Data Type: `EventType[]`. The list of `EventTypes` to be removed.
3837 Events of these types SHALL NOT be reported anymore.

3838

3839 **Return Value(s):**

3840 Data Type: `void`. This command SHALL NOT return a value.

3841

3842

3843 **6.9.9 DataSelector.removeAllEventFilters**

3844 Removes all event filters currently associated with this `DataSelector`.

3845 Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error
3846 (`ERROR_*_NOT_FOUND`).

3847

3848 **Compliance Requirement:** Compliant systems MAY implement this command.

3849

3850 **Usage:**

3851 `DataSelector.removeAllEventFilters (void): void`

3852

3853 **Parameter(s):**

3854 Data Type: `void`. This command takes no parameters.

3855

3856 **Return Value(s):**

3857 Data Type: `void`. This command SHALL NOT take any parameters.

3858

3859

3860 **6.9.10 DataSelector.getAllEventFilters**

3861 Returns all event filters currently associated with this `DataSelector`.

3862

3863 **Compliance Requirement:** Compliant systems MAY implement this command.

3864

3865 **Usage:**

3866 `DataSelector.getAllEventFilters (void): EventType[]`

3867

3868 **Parameter(s):**

3869 Data Type: `void`. This command takes no parameters.

3870

3871 **Return Value(s):**

3872 Data Type: `EventType[]`. A list of all `EventTypes` currently being reported.

3873

3874

3875

3876 **6.9.11 DataSelector.addTagFieldNames**

3877 Adds the specified tag field names to the list of tag field names currently associated with
3878 this `DataSelector`. If some of the given names to be added are already associated
3879 with this `DataSelector`, only the not yet associated names shall be added and the
3880 command SHALL complete successfully.

3881

3882 **Compliance Requirement:** Compliant systems MAY implement this command.

3883

3884 **Usage:**

3885 `DataSelector.addTagFieldNames (fieldNames: string[]): void`

3886

3887 **Parameter(s):**

3888 `fieldNames` – Data Type: `string[]`. The list of `TagField` names to be added.
3889 These data fields SHALL from now on be reported.

3890

3891 **Return Value(s):**

3892 Data Type: `void`. This command SHALL NOT return any value.

3893

3894

3895

6.9.12 DataSelector.removeTagFieldNames

Removes the specified tag field names from the list of tag field names currently associated with this `DataSelector`. If one or more of the names given are not currently associated with this `DataSelector`, these are ignored and all other names SHALL be removed and the command SHALL complete successfully.

Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error (ERROR_*_NOT_FOUND).

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
DataSelector.removeTagFieldNames (fieldNames: string[]): void
```

Parameter(s):

`fieldNames` – Data Type: `string[]`. The list of `TagField` names to be removed. These tag data fields SHALL NOT be reported anymore.

Return Value(s):

Data Type: `void`. This command SHALL NOT return any value.

6.9.13 DataSelector.removeAllTagFieldNames

Removes all tag field names currently associated with this `DataSelector`.

Removing a 'default' object changes the 'default' to *null*. Using a '*null*' object is an error (ERROR_*_NOT_FOUND).

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

```
DataSelector.removeAllTagFieldNames (void): void
```

Parameter(s):

Data Type: `void`. This command SHALL NOT take any parameters.

3930

3931 **Return Value(s):**

3932 Data Type: `void`. This command SHALL NOT return any value.

3933

3934

3935 **6.9.14 DataSelector.getAllTagFieldNames**

3936 Returns all tag field names currently associated with this `DataSelector`.

3937

3938 **Compliance Requirement:** Compliant systems MAY implement this command.

3939

3940 **Usage:**

3941 `DataSelector.getAllTagFieldNames (void): string[]`

3942

3943 **Parameter(s):**

3944 Data Type: `void`. This command SHALL NOT take any parameters.

3945

3946 **Return Value(s):**

3947 Data Type: `string[]`. The list of all `TagField` names currently being reported.

3948

3949

3950

3951 **6.9.15 DataSelector.getMaxNumberSupported**

3952 Returns the maximum number of `DataSelectors` this Reader supports. This is a
3953 static method.

3954

3955 **Compliance Requirement:** Compliant systems SHALL implement this command.

3956

3957 **Usage:**

3958 `static DataSelector.getMaxNumberSupported (): integer`

3959

3960 **Parameter(s):**

3961 Data Type: `void`. This command SHALL NOT take any parameters.

3962

3963 **Return Value(s):**

3964 Data Type: `integer` . The maximum number of DataSelectors the Reader supports, or
3965 -1 if unbounded (no limit).

3966

3967

3968

3969 **6.10 Enumeration Objects**

3970 **6.10.1 Object EventType**

3971 The enumeration object `EventType` lists all possible events as shown in the state
3972 diagram in section 4.2.1 and also explained in section 5.5.

3973 **6.10.1.1 EventType.getSupportedTypes**

3974 Returns a list of all supported `EventTypes`.

3975

3976 **Compliance Requirement:** Compliant systems SHALL implement this command.

3977

3978 **Usage:**

3979 `static EventType.getSupportedTypes(): EventType[]`

3980

3981 **Parameter(s):**

3982 Data Type: `void` . This command SHALL NOT take any parameters.

3983

3984 **Return Value(s):**

3985 Data Type: `EventType[]` . The list of `EventTypes` that this reader supports.

3986

3987

3988 **6.10.2 Object TriggerType**

3989 The enumeration object `TriggerType` lists all possible trigger types, as described in
3990 section 5.3.

3991 **6.10.2.1 TriggerType.getSupportedTypes**

3992 Returns a list of all supported `TriggerTypes`.

3993

3994 **Compliance Requirement:** Compliant systems SHALL implement this command.
3995
3996 **Usage:**
3997 `static TriggerType.getSupportedTypes(): TriggerType[]`
3998
3999 **Parameter(s):**
4000 Data Type: `void`. This command SHALL NOT take any parameters.
4001
4002 **Return Value(s):**
4003 Data Type: `TriggerType[]`. The list of `TriggerTypes` that this reader supports.
4004
4005
4006
4007 **6.10.3 Object FieldName**
4008 The enumeration object `FieldName` lists all possible field names used to define what
4009 data shall be reported, as described in chapter 5.7.
4010
4011 **6.10.3.1 FieldName.getSupportedNames**
4012 Returns a list of all supported `FieldNames`.
4013
4014 **Compliance Requirement:** Compliant systems SHALL implement this command.
4015
4016 **Usage:**
4017 `static FieldName.getSupportedNames(): FieldName[]`
4018
4019 **Parameter(s):**
4020 Data Type: `void`. This command SHALL NOT take any parameters.
4021
4022 **Return Value(s):**
4023 Data Type: `FieldName[]`. The list of `FieldNames` that this reader supports.
4024
4025

6.10.4 Object PredefinedTagFieldName

The enumeration object `PredefinedTagFieldName` defines a set of tag field names used for a predefined set of tag data fields.

The following table defines the names of all known predefined tag field names, describes their meaning, and lists the compliance requirement.

Data field	Compliance	Description
tagId	MAY	The ID of the tag
accessPassword	MAY	The password to access the tag
killPassword	MAY	The password to kill the tag
<Vendorname>:<Any>	MAY	Vendors MAY define additional fields. The name of these fields SHALL have the vendor name and a colon as prefix.

Table 12: Predefined tag field names

6.11 Object TagField

`TagField` objects are used to define individual tag data fields. A tag field specifies where the data field is located in tag memory. Further each tag field has a name that makes it possible to associate data fields with the same or similar data types. For example, two different tag types, using different tag RF protocols, may need to store the same type of application data. However, these associated data fields may exist at different locations on the two different tag types.

6.11.1 TagField.create

Create a `TagField` object with a given name. If a `TagField` object with the same name exists already, an error is returned. This is a static method.

The `TagField` SHALL implicitly be added to the list of all `TagFields` kept by the `ReaderDevice` object.

The newly created object has its standard properties (if supported) initialized as:

Property	Value
Name	name
TagFieldName	<instance/vendor dependent>
MemoryBank	<instance/vendor dependent>
Offset	<instance/vendor dependent>
Length	<instance/vendor dependent>

Table 13: Default properties for TagField object

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

`static TagField.create(name: string): TagField`

Parameter(s):

`name` – Data Type: `string`. The name of the object to be created.

Return Value(s):

Data Type: `TagField`. The newly created object.

6.11.2 TagField.getName

Returns the name of the given `TagField` object. This name should not be confused with the `TagFieldName` attribute of the object. A `TagField` object's name SHALL be unique whereas a `TagField` object's `TagFieldName` attribute MAY be shared (see 6.11.3 `TagField.getTagFieldName` for details).

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

`TagField.getName (void): string`

Parameter(s):

Data Type: `void`. This command SHALL NOT take any parameters.

Return Value(s):

Data Type: `string`. The name of the `TagField` object is returned.

6.11.3 TagField.getTagFieldName

Returns the tag field name of this TagField object. This TagFieldName attribute is not the same entity as the TagField object's name. Two or more TagField objects can share the same TagFieldName attributes. This may be necessary to associate tag data fields that carry the same type of application data but exist in different locations on different tag types (i.e., different tag RF protocols).

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

TagField.getTagFieldName(void): string

Parameter(s):

Data Type: void. This command SHALL NOT take any parameters.

Return Value(s):

Data Type: string. The tag field name for this TagField.

6.11.4 TagField.setTagFieldName

Sets the TagFieldName attribute for this TagField object (see 6.11.3 TagField.getTagFieldName for details).

Compliance Requirement: Compliant systems MAY implement this command.

Usage:

TagField.setTagFieldName(tagFieldName: string) : void

Parameter(s):

tagFieldName - Data Type: string. The tag field name for this TagField.

Return Value(s):

4113 Data Type: `void`. This command SHALL NOT return any value.

4114

4115

4116 **6.11.5 TagField.getMemoryBank**

4117 Returns the value corresponding to the `TagField` object's memory bank attribute. The
4118 application of this attribute is dependent upon a tag's RF protocol. The use of this
4119 attribute and its application to specific RF protocols SHALL be documented.

4120

4121 **Compliance Requirement:** Compliant systems MAY implement this command.

4122

4123 **Usage:**

4124 `TagField.getMemoryBank(void): integer`

4125

4126 **Parameter(s):**

4127 Data Type: `void`. This command SHALL NOT take any parameters.

4128

4129 **Return Value(s):**

4130 Data Type: `integer`. The memory bank for this `TagField`.

4131

4132

4133

4134

4135 **6.11.6 TagField.setMemoryBank**

4136 Sets the value corresponding to the `TagField` object's memory bank attribute. The
4137 application of this attribute is dependent upon a tag's RF protocol. The use of this
4138 attribute and its application to specific RF protocols SHALL be documented.

4139

4140 **Compliance Requirement:** Compliant systems MAY implement this command.

4141

4142 **Usage:**

4143 `TagField.setMemoryBank(memoryBank: integer) : void`

4144

4145 **Parameter(s):**

4146 memoryBank - Data Type: integer . The memory bank for this TagField.

4147

4148 **Return Value(s):**

4149 Data Type: void . This command SHALL NOT return any value.

4150

4151

4152 **6.11.7 TagField.getOffset**

4153 Returns the value corresponding to the TagField object's offset attribute. The exact
4154 application of this attribute is dependent upon a tag's RF protocol. However, the offset
4155 SHALL always be specified in units of memory *bits*. The use of this attribute and its
4156 application to specific RF protocols SHALL be documented.

4157

4158 **Compliance Requirement:** Compliant systems MAY implement this command.

4159

4160 **Usage:**

4161 TagField.getOffset(void): integer

4162

4163 **Parameter(s):**

4164 Data Type: void . This command SHALL NOT take any parameters.

4165

4166 **Return Value(s):**

4167 Data Type: integer . The offset for this TagField.

4168

4169

4170

4171

4172 **6.11.8 TagField.setOffset**

4173 Sets the value corresponding to the TagField object's offset attribute. The exact
4174 application of this attribute is dependent upon a tag's RF protocol. However, the offset
4175 SHALL always be specified in units of memory *bits*. The use of this attribute and its
4176 application to specific RF protocols SHALL be documented.

4177

4178 **Compliance Requirement:** Compliant systems MAY implement this command.

4179

4180 **Usage:**

4181 TagField.setOffset(offset: integer) : void

4182

4183 **Parameter(s):**

4184 offset - Data Type: integer . The offset for this TagField.

4185

4186 **Return Value(s):**

4187 Data Type: void . This command SHALL NOT return any value.

4188

4189

4190 **6.11.9 TagField.getLength**

4191 Returns the value corresponding to the TagField object's length attribute. The exact
4192 application of this attribute is dependent upon a tag's RF protocol. However, the length
4193 SHALL always be specified in units of memory *bits*. The use of this attribute and its
4194 application to specific RF protocols SHALL be documented.

4195

4196 **Compliance Requirement:** Compliant systems MAY implement this command.

4197

4198 **Usage:**

4199 TagField.getLength(void): integer

4200

4201 **Parameter(s):**

4202 Data Type: void . This command SHALL NOT take any parameters.

4203

4204 **Return Value(s):**

4205 Data Type: integer . The length for this TagField.

4206

4207

4208

4209 **6.11.10 TagField.setLength**

4210 Sets the value corresponding to the TagField object's length attribute. The exact
4211 application of this attribute is dependent upon a tag's RF protocol. However, the length

4212 SHALL always be specified in units of memory *bits*. The use of this attribute and its
4213 application to specific RF protocols SHALL be documented.

4214

4215 **Compliance Requirement:** Compliant systems MAY implement this command.

4216

4217 **Usage:**

4218 TagField.setLength(length: integer) : void

4219

4220 **Parameter(s):**

4221 length - Data Type: integer . The length for this TagField.

4222

4223 **Return Value(s):**

4224 Data Type: void . This command SHALL NOT return any value.

4225

4226

4227

4228 **6.11.11 TagField.getMaxNumberSupported**

4229 Returns the maximum number of TagFields this Reader supports. This is a static
4230 method.

4231

4232 **Compliance Requirement:** Compliant systems SHALL implement this command.

4233

4234 **Usage:**

4235 static Source.getMaxNumberSupported(): integer

4236

4237 **Parameter(s):**

4238 Data Type: void . This command SHALL NOT take any parameters.

4239

4240 **Return Value(s):**

4241 Data Type: integer . The maximum number of TagFields the Reader supports, or -1
4242 if unbounded (no limit).

4243

4244

7 Reader Layer – Notifications

This section specifies messages exchanged at the Reader Layer over the notification channel. As described in section 3, messages on the notification channel are sent asynchronously from Reader to Host. Some command messages sent on the control channel MAY serve as triggers for notification messages.

There are two types of notification messages:

- **Events**
- **MTB-specific notifications**

These notifications are described below using the same UML-like notation as defined in chapter 6.

7.1 Events

7.1.1 Read Report

A read report is a notification the reader sends to the host, informing it about tag reads in the report buffer. What data is reported inside a read report is defined by the configuration of the `DataSelector` object associated with the notification channel. The exact format of the report is defined in the MTB.

7.1.2 Vendor-defined Notification

A vendor MAY issue additional notifications. Vendor-specific notifications SHALL be marked as such and include the vendor name to disambiguate notifications from different vendors.

7.2 MTB-specific notification

This section includes notifications specific to communication and interaction with a reader.

7.2.1 Goodbye

A Goodbye message is sent indicating the conversation will be terminated. This notification is used to indicate a ‘graceful disconnect’ distinguishing proper system operation from a ‘disgraceful disconnect’ due to loss of network connectivity, unexpected shutdown of a Host or Reader, or other ‘surprising’ conditions.

This notification is equivalent to the Goodbye command sent on the control channel by the host to the reader, but in this case the reader is informing the host that it is going to cease operations.

4281

4282 **Compliance Requirement:** Compliant systems MAY implement this notification.

4283

4284 **Usage:**

4285 goodbye (void): void

4286

4287 **Parameter(s):**

4288 Data Type: void. This notification SHALL take no parameters.

4289

4290 **Return Value(s):**

4291 Data Type: void. This notification SHALL NOT return any value.

8 Error Handling

8.1 Error Conditions

There are several types of error conditions:

- Communication errors, either when a host issues a command or when the reader tries to send a notification (e.g., tag list report) to the host
- Command execution errors that occur as response to commands sent from the host to the reader.
- Vendor extensions

8.2 Communication Errors

8.2.1 Communication Host-to-Reader

These kinds of communication errors occur when the host tries to issue a command to the reader but the reader is not responding. These conditions are reported by the MTB-specific Transport Layer.

8.2.2 Communication Reader-to-Host

This specification does not define how the reader reacts when it isn't able to connect or exchange data over a notification channel; the exact behavior is implementation dependent. Notifications could be dropped, sent over a secondary channel, queued for later retry, raise a trap in a management console, and/or many other behaviors.

8.3 Command Errors

All commands support a standard error structure, for use in the event of errors. This contains standard fields (some required, some optional) as well as support for vendor extensions.

All standard error conditions defined can be reported with additional data about the error (formatted and transmitted as per the MTB in use):

- **SHALL:** The **error code** and/or **error name**. At least one of these fields SHALL be included in the **Error** structure. The exact contents depend on the binding, and SHALL be constant at the binding level, e.g. MTB X can be defined as always including the **error code** (but not the **error name**)
- **SHALL** (where applicable): Information about the **error cause**. For example, for input parameter errors, information on which parameter failed needs to be included. Depending on the binding, this can be either the name or the index of the parameter. Vendor extensions are allowed here.
- **SHALL** (where applicable): A **vendor name or identifier** for the responses to vendor-specific commands SHALL be given.
- **MAY:** A **descriptive text string**, mainly used for logging purposes. The language for the text is implementation-dependent and MAY be configurable.

4328 The error code is a 16-bit (2-byte) integer. The following table defines the mapping of
4329 error conditions / error codes.

4330

Error Code (Hex)	Error Name	Description
0000	NO_ERROR	
0001	ERROR_UNKNOWN	Unspecified error.
0002	ERROR_COMMAND_NOT_SUPPORTED	The command is not supported or unknown to the reader. Unsupported commands SHALL always return ERROR_COMMAND_NOT_SUPPORTED, regardless of the reason (whether spec'd but optional and not implemented -or- unknown/invalid).
0003	ERROR_PARAMETER_INVALID_FORMAT	A parameter has the wrong format. Examples: • <code>addTagSelector</code>: The mask or value parameters contain characters with no defined meaning • Trigger-value strings cannot be parsed correctly • <code>setTimeUTC</code>: time format
0004	ERROR_PARAMETER_MISSING	A required parameter is not specified.
0005	ERROR_PARAMETER_INVALID_DATATYPE	A parameter has the wrong format, e.g. string instead of an integer.
0006	ERROR_PARAMETER_ILLEGAL_VALUE	Using a parameter value that is illegal. Example: Setting the port to 10 in the <code>addReadTrigger</code> command when there is no port 10.
0007	ERROR_PARAMETER_OUT_OF_RANGE	A parameter is out of the valid range for this parameter, e.g. setting the duty cycle to value not between 0 and 100.
0008	ERROR_PARAMETER_NOT_SUPPORTED	A parameter is not supported by this reader, e.g. asking a reader that doesn't support any read triggers to add a read trigger with a value other than <code>none</code> .

Error Code (Hex)	Error Name	Description
0009	ERROR_PARAMETER_LENGTH_EXCEEDED	The length of the given parameter was too long, e.g. when providing a string value in a setAttribute method that is longer than the maximum length of that attribute.
000A	ERROR_TOO_MANY_TAGSELECTORS	Raised when trying to add more than the maximum number of tag selectors.
000B	ERROR_TOO_MANY_TRIGGERS	Raised when trying to add more than the maximum number of triggers.
000C	ERROR_TAGSELECTOR_NOT_FOUND	Raised when the given tag selector parameter is not known.
000D	ERROR_TRIGGER_NOT_FOUND	Raised when the given trigger parameter is not known.
000E	ERROR_READPOINT_NOT_FOUND	Raised when the given read point parameter is not known.
000F	ERROR_CHANNEL_NOT_FOUND	Raised when the given channel parameter is not known.
0010	ERROR_SOURCE_NOT_FOUND	Raised when the given source parameter is not known.
0011	ERROR_SOURCE_READ_ONLY	Raised when trying to remove a source that cannot be removed, e.g. if it is one of the preconfigured sources.
0012	ERROR_DATASELECTOR_NOT_FOUND	Raised when the given DataSelector parameter is not known.
0013	ERROR_TAGS_DETECTED_NOTREAD	Tags were detected by the reader, but cannot be read successfully. Note: 'No tags detected' is a valid result, not an error.
0014	ERROR_NO_TAG	No tag was in the field of the reader when trying to write or kill.
0015	ERROR_TAG_LOCKED	The to be written or killed is locked.
0016	ERROR_WRONG_PASSWORD	The given kill code is wrong.
0017	ERROR_MULTIPLE_TAGS	Multiple tags were detected in the range of the reader when trying to write or kill.
0018	ERROR_READONLY_TAG	The tag was a read-only tag that cannot be written.

Error Code (Hex)	Error Name	Description
0019	ERROR_OBJECT_EXISTS	Raised when trying to create an object and an object of the same type with the same name exists already.
001A	ERROR_READPOINT_IN_USE	The reader does not allow ReadPoints shared between Sources and the ReadPoint is already assigned to another Source.
001B	ERROR_TAGFIELD_NOT_FOUND	Raised when the given tag field parameter is not known.
001C	ERROR_PARSING_MESSAGE	Raised when an error occurs parsing a message (e.g. malformed message).
001D - FFFE		Reserved for future use
FFFF	ERROR_VENDOR_EXTENSION	A vendor-specific error. See the Error structure's vendor extension section for the (vendor specific) error code.

Table 14: Basic Error Codes

Note that if the values of several parameters of a command result in an error, only one of the above error conditions can be returned. Which one is chosen depends on the implementation and is not defined in this specification.

8.4 Vendor Extensions

Vendors SHALL use the error conditions defined above for the commands of the standard command set. Vendors MAY add additional information on the exact cause of the error.

Vendors can extend the standard Reader Protocol command set. These extension commands MAY use the same general error conditions as defined above, however they MAY also define additional error conditions. These vendor-defined error conditions SHALL use names starting with <VENDOR>_ERROR..., for example ACME_ERROR_WHATSIT_NOT_ADDED_NO_FLABOOZLE if Acme Corp. needed an error condition indicating there was no WHATSIT associated with a FLABOOZLE.

9 Message/Transport Bindings (MTBs)

MTBs perform the function of transporting the encoded Reader Protocol commands between Readers and Hosts. The MTB definitions consist of formats and procedures that define the following:

1. The handshake exchange that is used to negotiate the specific Message Layer encoding. This exchange occurs in a text-based format that SHALL be understood by all implementations, regardless of Reader Layer encoding support.
2. The underlying transport layer, TCP/IP, HTTP, or serial, and how the connections are established.
3. Any Reader Layer message payload transformation that occurs before messages are delivered to the Transport Layer.
4. The framing of Reader Layer message payloads for delivery to the Transport Layer; e.g., by adding a length indication, channel identifier, etc.

Two message formats (XML and Text [non-XML]) can be used over any of the 3 specified transports (Serial, TCP, and HTTP). In the future, additional message formats and transports may be defined.

9.1 Address Notation

All transports support a URI naming convention when an address is required by the command set, e.g. `NotificationChannel.setAddress()`. This uses the following notation except where explicitly stated otherwise:

`<transport>://<locator>[?mode=(connect|listen)]`

For example:

`tcp://foo.bar.com:2000`

`tcp://foo.bar.com?mode=listen`

`serial://com1:3`

`http://myreaderisawebserver:7183`

Note 1: in the case of a serial connection, where multiple channels have to be multiplexed over a single connection, the 'port' is a number that identifies the channel.

Note 2: currently, *mode* only applies to the TCP transport. If *mode=connect*, then the Reader SHALL initiate a connection to the Host. If *mode=listen*, then the Reader SHALL assign a port to the channel and listen for a connection on that port.

10 Transports

10.1 Handshaking

All transports use a common handshaking protocol upon connection, negotiating expected behavior for all future messages exchanged over this connection. This handshaking occurs immediately upon connection establishment prior to any other transmission over the transport protocol, and it applies to all uses of MTBs (control and notification channels). These behaviors can only be negotiated at connection establishment; a new connection is required to use different handshaking properties.

10.1.1 Terminology

The following terms are used in the description of the handshake syntax and operations.

- Sender – the originator of the connection request
- Receiver – the receiver of the connection request

As an example, a Host would be the Sender of the handshake when it initiates a control connection to a Reader. A Reader would be the Sender of the handshake if it initiates a data notification channel to a Host.

10.1.2 Syntax

10.1.2.1 Sender Handshake

The following are the fields in the Sender Handshake. The values for these fields are sent sequentially in ASCII printable characters. The currently defined operations for the protocol result in a 20 byte fixed length handshake format with the reserved field included for potential future extension.

Field Name	Length (characters)	Value(s)
sender-signature	4	“RPS1”
spec-version-request	2	“11”
sender-format-request	2	Message format (“B” “T” “X”) for binary, text, or XML) + message format version (“1”). In RP1.1, only “T1” and “X1” is supported.
Receiver-format-request	2	Message format (“B” “T” “X”) for binary, text, or XML) + message format version (“1”). In RP1.1, only “T1” and “X1” is supported.
Ack-nak-request	2	“AR” if specific acknowledgements for

		commands are requested, “NA” otherwise
sender-reserved	4	“0000”
trailer	4	“END1”

Table 15: Sender Handshake fields

10.1.2.2 Receiver Handshake

The following are the fields in the Receiver Handshake. The values for these fields are sent sequentially in ASCII printable characters. The currently defined operations for the protocol result in a 20 byte fixed length handshake format with the reserved field added for potential future extension.

Field Name	Length (characters)	Value(s)
receiver-signature	4	“RPR1”
response	2	“OK” if all requested capabilities supported, “NO” otherwise.
Spec-version-response	2	“11”. When future versions are supported, the response MAY be a different version if the requested version is not supported.
Sender-format-response	2	Requested message format and version if supported, “NO” otherwise.
Receiver-format-response	2	Requested message format and version if supported, “NO” otherwise.
Ack-nak-response	2	“OK” if ack-nak-request value supported, “NO” otherwise.
Receiver-reserved	2	“00”
trailer	4	“END1”

Table 16: Receiver Handshake fields

10.1.3 Operations

10.1.3.1 Initiation

Upon establishing a connection via the transport mechanism, the Sender SHALL send the Sender-Handshake. Upon establishing a connection via the transport mechanism, the Receiver SHALL read the Sender-Handshake and SHALL then write the Receiver-Handshake back to the Sender with values indicating the response to the requested

4418 support. The Receiver SHALL write the ReceiverHandshake before terminating the
4419 connection.
4420 If the Receiver reads an invalid sender-handshake it SHALL NOT respond and drop the
4421 connection. An 'invalid' sender-handshake is defined as a non-matching 'sender-signature'
4422 ("RSP1") or 'trailer' ("END1") field. This no-response behavior is a good security
4423 practice; the receiver-handshake SHALL only be transmitted if a recognizable sender-
4424 handshake is received.

4425 **10.1.3.2 Versioning and Capabilities Request**

4426 The parameters of spec-version-request, sender-format-request, receiver-format-request,
4427 and ack-nak-request MAY be interpreted as hints, not hard requirements. They indicate
4428 the operation relative to the specific parameter that the Sender can understand and
4429 expects to handle. If the Receiver is able to operate at the requested level or perform the
4430 requested function, the Receiver responds with the response value equal to the request
4431 value. Note that this does not necessarily mean that the Receiver is actually operating at
4432 the requested level; it just means that it is able to respond properly to commands, and that
4433 all required functionality and commands (i.e., those termed "SHALL") are supported.

4434

4435 In the case of a Receiver operating at a version different than the one indicated in the
4436 request, and the Receiver deciding that it is not able to operate at the required level, the
4437 Receiver SHALL respond with the value of "NO". The Sender can then decide whether
4438 to attempt to retry the connection request with a different set of parameters.

4439

4440 **10.1.3.3 Extensibility**

4441 Support for future extensibility is provided through the reserved field in both the Sender
4442 and Receiver Handshake formats. In Reader Protocol 1.1, this field SHALL be set to all
4443 0's when sent and SHALL be ignored upon receipt.

4444 **10.2 Serial**

4445 There are two serial binding modes defined:

- 4446 - human-to-machine (TTY)
- 4447 - machine-to-machine (M2M)

4448 A human-to-machine mode is provided for device demonstrations, prototyping,
4449 development, debugging, and maintenance. This mode minimizes user input
4450 requirements however it does not provide a robust, reliable interface. The machine-to-
4451 machine mode is intended for production environments where a robust interface is
4452 required.

10.2.1 Transport Address Notation

In accordance with section 9.1 *Address Notation*, the URI naming convention SHALL be used to address a serial communications transport. The *transport* of the URI is the keyword:

serial

The locator of the URI has the syntax:

com<physical port number>[':'<logical port number>]

Examples:

serial://com1

serial://com2:5

10.2.2 Serial Interface Physical Characteristics

A Reader SHALL come from a manufacturer with published default physical device characteristics including, but not limited to, the following items:

- Baud rate
- Data bits
- Stop bits
- Parity
- Flow control

The Host SHALL initially communicate on the Reader's serial interface using the published default device characteristic values. The Host can later negotiate new device characteristic values via reader configuration commands.

10.2.3 Human-to-Machine Mode (TTY)

This MTB carries messages over a serial interface using a lightweight framing protocol. A goal of this MTB is to permit a human interface to readers requiring minimal user input.

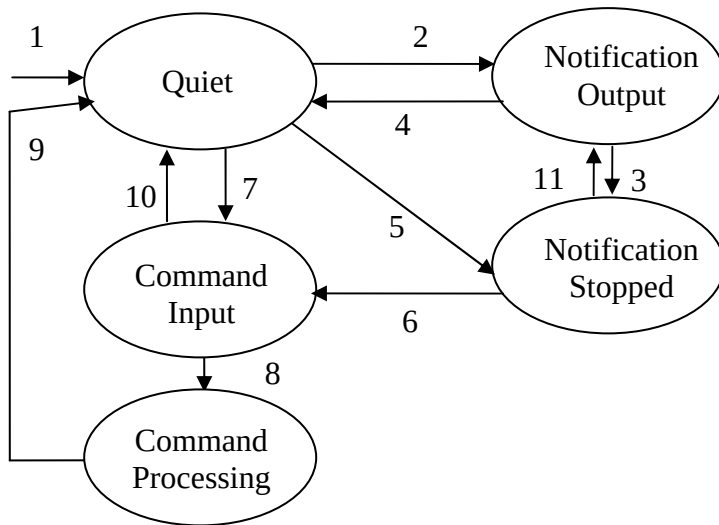


Figure 10: States of a serial transport connection

In this MTB, two connections are established; one to carry the Control Channel and another (initiated by the Reader) is used when needed to carry the Notification Channel. There are restrictions placed upon the Notification Channel to help insure that the output from this channel doesn't become interlaced with Control Channel data. While in TTY mode, the Reader SHALL maintain the following state machine.

The following list enumerates these states and describes the mechanics of each state. In the above state diagram each state transition is annotated by a number. Below, the description of each state is followed by a description of each state transition from the state. Start-up state transition #1 is described with the Quiet state.

- **Quiet**

The Reader can only report asynchronous data over the Notification Channel when the Reader is in the Quiet state. As it begins sending notification data, the Reader SHALL transition to the Notification Output state. From the Quiet state, a null command (i.e., a command terminator) SHALL cause the Reader to transition to the Notification Stopped state.

Whenever the Reader transitions to the Quiet state (including the initial entry), the Reader SHALL send a command prompt to the user.

1- The Reader SHALL enter the Quiet state initially.

2- If it begins sending notification data, the Reader SHALL transition to the Notification Output state.

5- If a null command is detected the Reader SHALL transition to the Notification Stopped state.

7- The Reader SHALL transition to the Command Input state whenever input is detected from the user other than a command terminator.

4506

4507

- **Command Input**

4508

4509

4510

4511

The Reader SHALL enter the Command Input state whenever a single input character is received from the user other than a command terminator. Once in the Command Input state, the Reader SHALL stay in this state until a command terminator is received or a timeout occurs.

4512

4513

8- When a command terminator is received the Reader SHALL transition to the Command Processing state.

4514

4515

4516

10- The Reader SHALL transition from the Command Input state back to the Quiet state if the command terminator is not received within a specified timeout period.

4517

4518

- **Command Processing**

4519

4520

The Reader SHALL enter the Command Processing state from the Command Input state once a command terminator is received

4521

4522

4523

4. The Reader SHALL stay in the Command Processing state until the last character of the command response has been transmitted to the user. The Reader SHALL then transition to the Quiet State.

4524

4525

- **Notification Output**

4526

4527

4528

4529

The Reader SHALL enter the Notification Output state from the Quiet state when it has asynchronous data to transmit on the Notification Channel. Once in the Notification Output state, the Reader SHALL ignore any user input received. Once notification data has completed, the Reader SHALL re-enter the Quiet state.

4530

4531

4532

4533

It is possible that the Reader MAY generate asynchronous notification data continuously. In this case the user MAY become locked out and unable to enter a command. To manage this situation, while in the Notification Output state, the Reader SHALL detect a null command.

4534

4535

4536

3- When a null command is detected the Reader SHALL interrupt the notification output at a reasonable breaking point (implementation dependent) and transition to the Notification Stopped state.

4537

4538

4- When notification data has completed, the Reader SHALL re-enter the Quiet state.

4539

4540

- **Notification Stopped**

4541

4542

The Reader SHALL transition to the Notification Stopped state from either the Quiet state or the Notification Output state whenever a null command is detected.

4543 While in the Notification Stopped state, the Reader SHALL hold all notification
4544 data until the Reader re-enters the Quiet state.

4545 The Reader SHALL transition from the Notification Stopped state to the
4546 Command Input state whenever a single input character is received other than a
4547 command terminator. While in the Notification Stopped state, if a command
4548 terminator is received, the Reader SHALL remain in the Notification Stopped
4549 state.

4550 6- When a single input character is received other than a command terminator the
4551 Reader SHALL transition to the Command Input state.

4552 11- Once in the Notification Stopped state for thirty (30) seconds or longer, the
4553 Reader SHALL transition to the Notification Output state where it will resume
4554 sending queued notification information.

4555 After a serial TTY connection is established, the handshake defined in section 10.1 MAY
4556 be performed to determine connection characteristics such as the message encoding for
4557 data transmitted over the connection. The specific encoding and message framing rules
4558 for each binding are then used for sending and receiving of the messages within the serial
4559 data stream. If the handshake is not performed, then the Reader SHALL use pre-defined
4560 encoding and message framing rules established for its serial TTY binding.

4561

4562 **10.2.3.1 Connection Establishment (Connect)**

4563 The Reader SHALL listen for an incoming serial request from the Host (Control
4564 Channel) on a pre-configured serial I/O communications port.

4565 **10.2.3.1.1 Control Channel**

4566 When a user wants to establish a TTY serial connection with a Reader, the user MAY
4567 send the handshake request (see section 10.1) or the user SHALL send a connection
4568 initialization message which consists of the ASCII text:

4569 TTY<CommandTerminator>

4570 Where 'TTY' is literal but it is case-insensitive. A command terminator is one of the
4571 following ASCII text patterns:

4572 CommandTerminator → <carriage return> <line feed>

4573 CommandTerminator → <line feed>

4574 These command terminator patterns are designed to be compatible with all known TTY
4575 simulation software utilities.

4576 When a Reader receives a TTY connection request, it SHALL accept or reject the request
4577 by sending the appropriate Receiver-Handshake response that SHALL be framed as
4578 follows:

4579 <handshake reply><EOLTerminator>

4580 See section 10.1.2.2 for the handshake reply. The end-of-line (EOL) terminator is
4581 defined as the following ASCII text sequence:
4582 EOLTerminator → <carriage return><line feed>
4583 When a Reader accepts a TTY request on the Control Channel, the Reader SHALL refuse
4584 subsequent requests for connection on that port as long as the TTY connection between
4585 Reader and Host exists on that port.
4586 The connection established by this method persists until it is explicitly torn down (see
4587 Section 10.1.2) or is lost due to Reader, Host or network failure.

4588 **10.2.3.1.2 Notification Channel**

4589 The Reader SHALL send notification information following the rules established by the
4590 state machine described in section 10.2.3. The Reader SHALL wait for the Control
4591 Channel to be quiet before transmitting Notification Channel data. In this state, the
4592 Reader can send asynchronous messages to the user. All Notification Channel messages
4593 SHALL be prefixed with the text:

4594 NOTIFICATION:<EOLTerminator>

4595 And, each message SHALL also be suffixed with an EOLTerminator (see section
4596 10.2.3.1.1 above) i.e.,

4597 <notification message><EOLTerminator>

4598 Any user input detected while a notification message is being written SHALL be
4599 discarded by the Reader (see the state machine description in section 10.2.3 above). The
4600 user SHALL be obliged to re-enter discarded data. This is necessary to insure that the
4601 user's command line maintains continuity. Note that there will usually not be more than a
4602 few characters discarded because the Reader waited for a quiet state before transmitting a
4603 notification message.

4604 **10.2.3.2 Connection Termination (Disconnect)**

4605 The user or the Reader can terminate the control channel as described in the following
4606 two subsections.

4607 **10.2.3.2.1 Control Channel**

4608 The Reader can terminate the Control Channel based upon a user inactivity timeout or
4609 data errors. This mechanism is implementation dependent. If the Reader terminates the
4610 Control Channel it SHALL send a termination notification to the user:

4611 <EOLTerminator> (see section 10.2.3.1.1 above)

4612 TERMINATED<EOLTerminator>

4613 The user can terminate a channel connection by issuing the ReaderDevice.goodbye
4614 command.

4615 **10.2.3.2.2 Notification Channel**

The TTY mode Notification Channel is automatically terminated when the TTY mode Control Channel is terminated. Also, the Notification Channel can be terminated by a one of the commands `ReaderDevice.removeNotificationChannel` or `ReaderDevice.removeAllNotificationChannels`. The Reader SHALL NOT otherwise terminate the Notification Channel.

10.2.3.3 Message Format

The message format used for sending and receiving data is defined by the specific message encoding negotiated. See section 11. [Message Formats](#) for more details.

10.2.4 Machine-to-Machine Mode (M2M)

This MTB carries messages over a serial interface using a lightweight framing protocol. A goal of this MTB is to permit a reliable connection between readers and host systems. Note that, unlike TCP, a serial interface does not innately include a mechanism to detect and correct data communication errors. Therefore, this serial M2M mode binding defines a transport layer just below the Reader Protocol's command/response layer for detecting and correcting data errors.

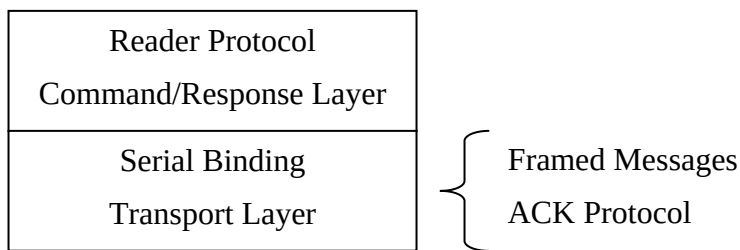


Figure 11: TCP Machine-to-Machine mode

Note that existing serial protocols were studied to determine whether an existing serial protocol could be applied here. It was discovered that existing serial protocols are file transfer-oriented and that they therefore include complexity not required for Reader Protocol. Therefore, salient features of existing serial protocols have been applied here to produce a minimal reliable message transfer-oriented serial protocol.

In this MTB, two connections are established; one to carry the Control Channel and another (initiated by the Reader) is used when needed to carry the Notification Channel.

The serial M2M mode transport layer requires that all messages sent by either the Host or the Reader SHALL be framed as follows:

<Header><message content><Trailer>

The purpose of the *header* and *trailer* is to detect data communications errors and to accommodate data recovery. Header and trailer fields include the following sub-fields:

Header → <Length><Message-Type><Port-Number><Message-ID>

4648 Trailer → <Checksum>

4649

4650 The sub-fields are defined as:

4651

4652 Length → four ASCII hexadecimal characters (the length represents the entire
4653 Reader frame size (number of bytes in hexadecimal) including the Port-Number
4654 and the Trailer fields. Length does not include the four bytes of the Length field.

4655 Message-Type → one ASCII character establishing message type:

4656 A – acknowledgement

4657 D – command

4658 R – response

4659 N – notification

4660

4661 Port-Number → two ASCII hexadecimal characters

4662 The *port number* is always established by the host. The Reader
4663 documentation SHALL include information about the port(s) a reader is
4664 listening to by default for incoming requests. During a connection
4665 handshake initiated by the Host, the Host SHALL establish a new, unique
4666 port number. If the connection is established by the Reader, the Host
4667 SHALL first issue a `NotificationChannel.setAddress`
4668 command which defines a unique port number to be used for the
4669 corresponding channel. When the Reader connects it, SHALL use the port
4670 number established by the `setAddress` command.

4671

4672 Message-ID → two ASCII hexadecimal characters

4673 The message identifier is always established by the sender. The receiver
4674 SHALL acknowledge a message received using the sender's message
4675 identifier. If the sender resends a message (i.e., no ACK is received), then
4676 the sender SHALL reuse the same message identifier every time the
4677 message is resent until the message is acknowledged.

4678

4679 Checksum → four ASCII hexadecimal characters representing a checksum value.

4680 The checksum is calculated using the entire frame excluding this
4681 checksum field. The checksum value is defined by the CCITT standard
4682 16-bit CRC.

4683

4684 The serial M2M mode transport includes a message ACK protocol in addition to the
4685 Reader's command/response protocol. The purpose of the ACK protocol is to
4686 accomplish data error detection and data recovery. All messages sent by either the Host
4687 or the Reader SHALL be acknowledged by the receiver with the following text:

4688 <Length>A<Port-Number><Message-ID><Checksum>

4689 Where 'A' is the literal representation of an ACK message type. The ACK message
4690 header includes the channel and message identifiers that SHALL match the sender's
4691 identifiers. The sender SHALL wait for this ACK before proceeding. The sender MAY
4692 have a configurable timeout that establishes the maximum time it waits for an ACK. The
4693 receiver MAY also have a configurable timeout that establishes the maximum time the
4694 receiver SHALL wait to receive the end of a message.

4695 If the sender times out waiting for an ACK, the sender SHALL re-send the message.
4696 When re-sending a message, the sender SHALL reuse the same message identifier every
4697 time the message is resent until the message is acknowledged.

4698 If the receiver times out waiting to receive the end of a message, or the receiver cannot
4699 verify a frame's check sum, then the receiver SHALL NOT reply with an ACK.

4700 The Reader MAY have a retry limit or it MAY simply keep retrying indefinitely.
4701 However, the sender SHALL NOT proceed to send another message until the last
4702 message sent has been acknowledged.

4703 **10.2.4.1 Connection Establishment (Connect)**

4704 The Reader SHALL listen for an incoming serial request from the Host (Control
4705 Channel) on a pre-configured serial I/O communications port. A handshake protocol
4706 SHALL be used to establish serial connections. Just as for all other M2M messages,
4707 handshake messages are framed:

4708 <Header><handshake message><Trailer>

4709 Where *header* and *trailer* fields are defined in section 10.2.4 above.

4710 **10.2.4.1.1 Control Channel**

4711 When a Host wants to establish a M2M serial connection with a Reader, the Host SHALL
4712 send a connection initialization message (handshake) defined for the in section 10.1.2.1.
4713 The Reader SHALL respond with the corresponding handshake response message (also
4714 defined in section 10.1.2.2).

4715 When a Reader accepts a serial request on the Control Channel, the Reader MAY refuse
4716 subsequent requests for connection at that port as long as the serial connection between
4717 Reader and Host exists at that port. If a Reader chooses to accept more than a single
4718 control connection, the Reader is responsible for responding to commands on the channel
4719 from which the command is received.

4720 When a Reader receives a serial connection request, it SHALL accept or reject the
4721 request by sending the appropriate Receiver-Handshake with the appropriate responses as
4722 defined in Section 10.1.2.

4723 As specified in section 10.2.4 above, these messages SHALL be encapsulated within a
4724 header and trailer. And, each message SHALL be acknowledged in addition to the
4725 appropriate message response. i.e.:

4726 Host: <Header><serial handshake request><Trailer>

4727 Reader: ACK

4728 Reader: <Header><serial handshake reply><Trailer>

4729 Host: ACK

4730 Once the Reader's handshake reply is returned by the Host, the Header and Trailer
4731 formats MAY change if the handshake exchange has negotiated binary formatted
4732 messages. Otherwise, for text and XML formatted messages, the Header and Trailer
4733 keep the same text format.

4734 The connection established by this method persists until it is explicitly torn down (see
4735 Section 10.1.2) or is lost due to Reader, Host or network failure.

4736 **10.2.4.1.2 Notification Channel**

4737 The Host SHALL listen for an incoming serial request from the Reader (Notification
4738 Channel) on a pre-configured serial I/O communications port.

4739 When a Reader wants to establish a serial M2M mode Notification Channel connection
4740 with a Host it follows the same protocol as described in the control channel with one
4741 exception. Reader and Host are transposed, i.e. the Reader becomes the sender and the
4742 Host becomes the receiver. i.e.:

4743 Reader: <Header><serial handshake request><Trailer>

4744 Host: <Header>ACK

4745 Host: <Header><serial handshake reply><Trailer>

4746 Reader: <Header>ACK

4747 The header associated with the Reader's handshake reflect the port number specified by
4748 the Host when it issued a SetNotifyAddress for the corresponding channel. Both the
4749 Host and the Reader SHALL use this port number in the header of any subsequent
4750 messages exchanged on the notification channel established.

4751 **10.2.4.2 Connection Termination (Disconnect)**

4752 Either Host or Reader can terminate either the their control channel or notification
4753 channel connection. respectively, via the Goodbye command. This command is specified
4754 in the reader command reference.

4755 **10.2.4.2.1 Control Channel**

4756 The Either the Host or Reader can terminate the Control Channel based upon a useran
4757 inactivity timeout or , data errors. This mechanism, management functions or other
4758 reasons. The format is implementationencoding dependent. If the Reader terminates the
4759 Control Channel it SHALL send a termination notification to the Host as specified in the

4760 reader command reference. If the termination is indicated through notification then no
4761 further communication SHALL be expected from the reader after the notification record.

4762 The only result of a request for termination of a control connection is for the release of
4763 the serial connection through a communications port close sequence.

4764 **10.2.4.2.2 Notification Channel**

4765 Any M2M mode Notification Channel are automatically terminated when the M2M mode
4766 Control Channel is terminated. Also, the Notification Channel can be terminated by a
4767 user command ReaderDevice.removeNotificationChannel.

4768 **10.2.4.3 Message Format**

4769 The serial M2M mode messages are framed as follows (see section 10.2.4 above):

4770 Sender: <Header><message content><Trailer>

4771 Receiver: ACK

4772 The message format used for sending and receiving data is defined by the specific
4773 message encoding negotiated. See section 11. [Message Formats](#) for more details.

4774 **10.3 TCP**

4775 This transport provides a reliable connection between Readers and Host systems across
4776 IP networks. In this transport, two types of connections can be established; one initiated
4777 by a Host called the Control Channel and another initiated by either a Host or Reader as
4778 needed called the Notification Channel.

4779

4780 After a TCP connection is established, the handshake defined in section 10.1 is performed
4781 to determine connection characteristics such as the message encoding for data transmitted
4782 over the connection. The specific encoding and message framing rules for each binding
4783 are then used for sending and receiving of the messages within the TCP data stream.

4784 **10.3.1 Transport Address Notation**

4785 In accordance with section 9.1 *Address Notation*, the URI naming convention SHALL be
4786 used to address a TCP communications transport. The *transport* of the URI is the
4787 keyword:

4788 **tcp**

4789 The locator of the URI has the syntax:

4790 **<host address>[':<logical port number>']**

4791 Examples:

4792 **tcp://myHost**

4793 **tcp://136.179.128.1**

4794 **tcp://myHost:2001**

10.3.2 Connection Establishment (Connect)

A Reader SHALL listen for an incoming TCP request from a Host (Control Channel) on the RP_PORT. Other ports MAY be supported but such support is outside the scope of this specification. Also, if commanded by a Host, a Reader MAY listen for an incoming TCP request from a Host to establish a Notification Channel connection. In this case the Reader SHALL assign a port for listening.

A Host MAY listen for an incoming TCP request from a Reader (Notification Channel) on the RP_PORT. Other ports MAY be supported but such support is outside the scope of this specification.

The default value for RP_PORT, 4684, is registered with IANA (Internet Assigned Numbers Authority). See <http://www.iana.org/assignments/port-numbers> for more details.

10.3.2.1 Control Channel

When a Host wants to establish a TCP connection with a Reader, it initiates a TCP connection request to the RP_PORT at the Reader IP address. After successful connection, the Host sends the Sender-Handshake as defined in Section 10.1.2 with the desired message format and the supported options.

When a Reader receives a TCP connection request, it SHALL accept or reject the request by sending the appropriate Receiver-Handshake with the appropriate responses as defined in Section 10.1.2.

When a Reader accepts a TCP request on the Control Channel, the Reader MAY refuse subsequent requests for connection at that port as long as the TCP connection between Reader and Host exists at that port. If a Reader chooses to accept more than a single control connection, the Reader is responsible for responding to commands on the channel from which the command is received.

The connection established by this method persists until it is explicitly torn down (see Section 10.1.2) or is lost due to Reader, Host or network failure.

10.3.2.2 Notification Channel

A Host MAY command a Reader to provide a Notification Channel. The Host MAY request one of two Notification Channel modes: *connect* or *listen*. In connect mode the Reader SHALL initiate a connection to the Host. In listen mode the Reader SHALL listen for a connection initiated by the Host.

When a Reader wants to establish a TCP connection with a Host for a Notification Channel (i.e., connect mode was requested by the Host), the Reader initiates a TCP connection request to the RP_PORT to the Host IP address. After successful connection, the Reader sends the Sender-Handshake as defined in Section 10.1.2 with the desired message format and the supported options.

4835 When a Host receives a TCP connection request, it SHALL accept or reject the request
4836 by sending the appropriate Receiver-Handshake with the appropriate responses as defined
4837 in Section 10.1.2.

4838 When a Host wants to establish a TCP connection with a Reader for a Notification
4839 Channel (i.e., listen mode was requested by the Host), the Host must first determine the
4840 port number assigned by the Reader for the Notification Channel. The Host can retrieve
4841 this information from the reply to either Notification Channel commands `setAddress`
4842 or `getEffectiveAddress`. Given the correct port number, the Host initiates a TCP
4843 connection request to the Reader. After successful connection, the Host sends the Sender-
4844 Handshake as defined in Section 10.1.2 with the desired message format and the
4845 supported options.

4846 When a Reader receives a TCP connection request, it SHALL accept or reject the request
4847 by sending the appropriate Receiver-Handshake with the appropriate responses as defined
4848 in Section 10.1.2.

4849 The connection established by either of these two methods (i.e., connect or listen mode)
4850 persists at least until the Reader has sent its Notification message, at which point it MAY
4851 be closed. A Reader MAY choose to keep the connection open to be used for subsequent
4852 notifications.

4853 **10.3.3 Connection Termination (Disconnect)**

4854 Either Host or Reader can terminate their control channel or notification channel
4855 connection respectively, via the Goodbye command. This command is specified in the
4856 reader command reference.

4857 **10.3.3.1 Control Channel**

4858 Either the Host or Reader can terminate the Control Channel based upon an inactivity
4859 timeout, data errors, management functions or other reasons. The format is encoding
4860 dependent. If the termination is indicated through notification then no further
4861 communication SHALL be expected from the reader after the notification record.

4862

4863 The only result of a request for termination of a control connection is for the release of
4864 the TCP connection, either through a completed TCP close sequence or through the TCP
4865 connection timeout mechanisms.

4866 **10.3.3.2 Notification Channel**

4867 A notification channel to a Host is normally terminated (i.e., no network failure) when the
4868 Host calls the command `ReaderDevice.removeNotificationChannel` or
4869 `ReaderDevice.removeAllNotificationChannels`. The Reader MAY also
4870 send a goodbye notification and close the connection independently.

10.3.4 Message Format

The message format used for sending and receiving data is defined by the specific message encoding negotiated embedded within the standard framing for messages transmitted via this protocol.

Messages are composed of a header and body, where the header is transport-specific information required by this transport and the body is the information being exchanged (e.g. commands or responses or notifications).

The transport uses a length-prefix style framing, with optional chunking.

All transmissions (in either direction) require the following framing:

message = header body

header = [not-last-chunk-indicator] <payload-length> <end-of-header>

not-last-chunk-indicator = "+"

payload-length = dec-val

dec-val = (DIGIT)+

end-of-header = ":"

body = <any>

The **body** is the standard syntax as per the message format in use (e.g. Text or XML).

See section 11. [Message Formats](#) for more details.

The **not-last-chunk-indicator** indicates this is part of a message and additional packet(s) are required to receive the full message. This provides an efficient means for streaming messages, especially large messages that would otherwise require significant buffering.

The **payload-length** indicates number of bytes in the body. Leading zeroes are not permitted; **148** is valid but **0148** and **00000148** are not.

Examples

14:getAllSources LF

40:Source#DockDoor5.ReadIDs myDataSelector LF

The command `getAllSources` could be transmitted as

14:getAllSources LF

or

+3:get

11:AllSources LF

or

+3:get

4906 **+3:All**
4907 **8:Sources LF**
4908 or any equivalent thereof.

4909 **10.4 HTTP**

4910 This transport provides a reliable connection between Readers and Host systems using
4911 standard HTTP syntax and semantics, as per the HTTP 1.1 protocol [RFC2616]. A goal
4912 of this MTB is to permit HTTP-compatible network elements (proxies, gateways, tunnels,
4913 etc.) to be used.

4914 In this transport, one HTTP connection (initiated by the Host) is used to carry the Control
4915 Channel, while another HTTP connection (initiated by the Reader) is used when needed
4916 to carry the Notification Channel.

4917 The syntax and structure used by this transport differs from most wire protocols as HTTP
4918 is inherently stateless. All standard information and semantics are honored, though
4919 differently from stateful communication protocols to accommodate the nature of HTTP-
4920 based communications.

4921 As this MTB only provides for a stateless communication model³, where a connection
4922 may last only for the duration of one HTTP request/response pair, the HTTP transport
4923 works as Request/Response pairs. The underlying connection may be kept open across
4924 multiple exchanges as per RFC2616 (HTTP1.1), but the properties defined for an HTTP
4925 Request only apply to its corresponding HTTP Response; each HTTP Request may have
4926 different handshake values.. The remainder of this chapter refers to an HTTP Response in
4927 relation to its corresponding HTTP Request.

4928 Given the stateless nature of HTTP, the standard handshake is incorporated into the
4929 HTTP Request and Response via HTTP headers (where necessary). To further facilitate
4930 use of this transport using commonly accepted practices, most of these headers are
4931 optional whose absence may be ignored or inferred. Lastly, the **goodbye** command is
4932 useless, but a vendor MAY still choose to implement it. These behaviors are defined in
4933 more detail.

4934 Of significant note, the body of the HTTP Request and Response are standard messages
4935 are per the defined message formats – there are no additional or special leading or trailing
4936 bytes in the body of the HTTP streams.

4937 Readers and Hosts MAY support HTTPS; the rules for HTTP and HTTPS are identical,
4938 except for the protocol specified in a URL (http://... Vs. https://...) . If Readers and Hosts
4939 support HTTPS, they SHALL use HTTP Over TLS as defined in RFC 2818. If used,
4940 TLS SHALL be implemented as defined in RFC 2246 except that the mandatory cipher
4941 suite is TLS_RSA_WITH_AES_128_CBC_SHA, as defined in RFC 3268 with

³ The Reader is a stateful device but the command channel has no overt state associated with it. There is no 'cookie' or 'handle' or other 'state' of information the Host can create or reference. Multiple Hosts communicating with a Reader are sharing the Reader's state, with all the connotations of that state of affairs.

4942 CompressionMethod.null. Implementations MAY support additional cipher suites and
4943 compression algorithms as desired.

4944 **10.4.1 Transport Address Notation**

4945 In accordance with section 9.1 *Address Notation*, the URI naming convention SHALL be
4946 used to address a serial communications transport. The *transport* of the URI is the
4947 keyword:

4948 **http**

4949 The locator of the URI has the syntax:

4950 **<host address>[':'<logical port number>]**

4951 Examples:

4952 **http://myHost**

4953 **http://136.179.128.1**

4954 **http://myHost:2001**

4955

4956 **10.4.2 Connection Establishment (Connect)**

4957 All HTTP messages SHALL obey the following rules:

- 4958 • HTTP 1.1 is required
- 4959 • All standard RP headers named 'RP- *'
- 4960 • HTTP Header 'Content-Type' is required
- 4961 • HTTP Header 'Content-Length' is required
- 4962 • Additional HTTP headers MAY be present

4963 All HTTP Requests SHALL also obey the following rules:

- 4964 • Default URL
- 4965 • Method = POST
- 4966 • HTTP Header 'RP-Sender-Signature' is optional
- 4967 • HTTP Header 'RP-Spec-Version' is optional
- 4968 • HTTP Header 'RP-Response-Content-Type' is optional
- 4969 • HTTP Header 'RP-Response-ACKNAK' is optional

4970 All HTTP Responses SHALL also obey the following rules:

- 4971 • Status Code is required
- 4972 • HTTP Header 'RP-Receiver-Signature' is optional
- 4973 • HTTP Header 'RP-Spec-Version-OK' is optional

4974 • HTTP Header 'RP-Request-Content-Type-OK' is optional
4975 • HTTP Header 'RP-Response-Content-Type-OK' is optional
4976 • HTTP Header 'RP-Response-ACKNAK-OK' is optional
4977 To wit, the standard Handshaking fields (see section 10.1) are specified via RP-specific
4978 HTTP Headers, except the Content-Type header which specifies the format of the body
4979 (and Content-Length specifying the size of the body).
4980 These rules are true for all HTTP-based communication, regardless of direction or
4981 channel. These rules are explicitly defined in the following sections.

4982 **10.4.2.1 HTTP 1.1 is required**
4983 Only HTTP 1.1 is supported; HTTP 1.0 is not allowed.

4984 **10.4.2.2 All standard RP headers named 'RP-***
4985 Headers defined by this standard are named with the prefix 'RP-', for example 'RP-Spec-
4986 Version'. Vendor extensions SHALL NOT use this prefix for vendor extension header
4987 fields.

4988 **10.4.2.3 HTTP Header 'Content-Type' is required**
4989 The Content-Type header field SHALL always be present, as per the message format in
4990 the body of the request/response:

4991 • **Content-Type: application/octet-stream**
4992 Binary Message Format, as per handshake <sender-format> = 'B1'
4993 • **Content-Type: text/plain; charset=utf-8**
4994 Text Message Format, as per handshake <sender-format> = 'T1'
4995 • **Content-Type: text/xml; charset=utf-8**
4996 XML Message Format, as per handshake <sender-format> = 'X1'
4997 NOTE: RP 1.1 only supports the Text and XML Message Formats.
4998 See section 10.1 for more details on the the standard Handshaking mechanism.

4999 **10.4.2.4 HTTP Header 'Content-Length' is required**
5000 The Content-Length header field SHALL always be present, as per HTTP 1.1. See
5001 RFC2616 for more details.

5002 **10.4.2.5 Additional HTTP headers MAY be present**
5003 Optional RP headers and vendor extension headers may be present (as well as standard
5004 HTTP 1.1 defined headers).

5005 **10.4.2.6 HTTP Request**
5006 The following rules apply to HTTP Requests.

5007 **10.4.2.6.1 Default URL**

5008 Readers and Hosts SHALL, by default, support the URL
5009 Control Channel = `http://server:port/control`
5010 where *server* is the Reader's address and *port* the TCP/IP port on which the reader is
5011 listening, as per standard HTTP rules. The default port number SHALL be noted in the
5012 documentation of the Reader. Other resources may be supported but such support is
5013 outside the scope of this specification.

5014 **10.4.2.6.2 Method = POST**
5015 All HTTP requests use the POST method.
5016 NOTE: 'form-data' is NOT used. HTTP Request and Response bodies are straight byte
5017 streams containing the appropriate message format.

5018 **10.4.2.6.3 HTTP Header 'RP-Sender-Signature' is optional**
5019 This header is optional in an HTTP Request; if present, it indicates the signature of the
5020 handshake. For example
5021 **RP-Sender-Signature: RPS1**
5022 This corresponds to the handshake's <sender-signature> field.
5023 If not present, this header is assumed to have a value of 'RPS1'.
5024 See section 10.1 for more details on the the standard Handshaking mechanism.

5025 **10.4.2.6.4 HTTP Header 'RP-Spec-Version' is optional**
5026 This header is optional. If present, this header has the format:
5027 **RP-Spec-Version: 1.1**
5028 If not present, this header is assumed to have a value of '1.1'.
5029 This header corresponds to the handshake's <spec-version> field. See section 10.1 for
5030 more details on the the standard Handshaking mechanism.
5031

5032 **10.4.2.6.5 HTTP Header 'RP-Response-Content-Type' is optional**
5033 This header is optional. If present, this header has the format:
5034 **RP-Response-Content-Type : <content-type>**
5035 where <content-type> is a known message format (e.g. 'B1', 'T1' or 'X1').
5036 If not present, this header is assumed to have a value as per the HTTP Request's Content-
5037 Type.
5038 This header corresponds to the handshake's <receiver-format> field. See section 10.1 for
5039 more details on the the standard Handshaking mechanism.
5040

10.4.2.6.6 HTTP Header 'RP-Response-ACKNAK' is optional

This header is optional. If present, this header has the format:

RP-ACKNAK: <boolean>

where <boolean> is 'yes' or 'no'.

If 'no', the response's body SHALL be empty (i.e. Content-Length: 0). *NOTE:* The HTTP Response is always required; this header merely controls whether or not the response has a body (it always has a header).

If not present, this header is assumed to have a value of 'yes'.

This header corresponds to the handshake's <acknak> field. See See section 10.1 for more details on the the standard Handshaking mechanism.

10.4.2.7 HTTP Response

The following rules apply to HTTP Responses.

10.4.2.7.1 Status Code is required

All HTTP responses SHALL contain a status code, as per "6.1.1. Status Code and Reason Phrase" and "10. Status Code Definitions" in the HTTP 1.1 standard [RFC2616]. The status code is used to indicate communicate or handshake errors; command set errors SHALL NOT use the status code to convey this information. For example, if an unsupported command is received by a reader the HTTP Response SHALL contain a status code of 200 (OK) with the body containing a standard error response containing an error code of ERROR_COMMAND_NOT_SUPPORTED, formatted as per the appropriate message format.

10.4.2.7.2 HTTP Header 'RP-Receiver-Signature' is optional

This header is optional in an HTTP Response; if present, it indicates the signature of the handshake. For example

RP-Sender-Signature: RPR1

This corresponds to the handshake's <receiver-signature> field.

If not present, this header is assumed to have a value of 'RPR1'.

See section 10.1 for more details on the the standard Handshaking mechanism.

10.4.2.7.3 HTTP Header 'RP-Spec-Version-OK' is optional

This header is required if the HTTP Request contained the header 'RP-Spec-Version'. If present, this header has the format:

RP-Spec-Version-OK: <boolean>

where <boolean> is 'yes' or 'no'.

If not present, this header is assumed to have a value of 'yes'.

5076 This header corresponds to the handshake's <spec-version-ok> field. See section 10.1 for
5077 more details on the the standard Handshaking mechanism.

5078

5079 **10.4.2.7.4 HTTP Header 'RP-Request-Content-Type-OK' is optional**

5080 This header is required if the HTTP Request contained the header 'RP-Request-Content-
5081 Type'. If present, this header has the format:

5082 RP-Request-Content-Type: <boolean>

5083 where <boolean> is 'yes' or 'no'.

5084 If not present, this header is assumed to have a value of 'yes'.

5085 This header corresponds to the handshake's <sender-format-ok> field. See See section
5086 10.1 for more details on the the standard Handshaking mechanism.

5087

5088 **10.4.2.7.5 HTTP Header 'RP-Response-Content-Type-OK' is optional**

5089 This header is required if the HTTP Request contained the header 'RP-Response-
5090 Content-Type'. If present, this header has the format:

5091 RP-Request-Content-Type: <boolean>

5092 where <boolean> is 'yes' or 'no'.

5093 If not present, this header is assumed to have a value of 'yes'.

5094 This header corresponds to the handshake's <receiver-format-ok> field. See section 10.1
5095 for more details on the the standard Handshaking mechanism.

5096

5097 **10.4.2.7.6 HTTP Header 'RP-Response-ACKNAK-OK' is optional**

5098 This header is required if the HTTP Request contained the header 'RP-ACKNAK'. If
5099 present, this header has the format:

5100 RP-ACKNAK-OK: <boolean>

5101 where <boolean> is 'yes' or 'no'.

5102 If not present, this header is assumed to have a value of 'yes'.

5103 This header corresponds to the handshake's <acknak-ok> field. See section 10.1 for more
5104 details on the the standard Handshaking mechanism.

5105

5106 **10.4.2.8 Control Channel**

5107 When a Host wants to establish an HTTP connection with a Reader, it initiates an HTTP
5108 connection to the appropriate URL. After successful connection, the Host sends an HTTP
5109 Request.

5110 When a Reader receives an HTTP request, it SHALL accept or reject the request by
5111 sending an HTTP Response with the appropriate structure (headers, body, etc.) as defined
5112 by this transport.

5113 When a Reader accepts an HTTP request on the Control Channel, the Reader MAY
5114 refuse subsequent requests for connection at that URL as long as the HTTP request is
5115 being processed. If a Reader chooses to accept more than a single control request, the
5116 Reader is responsible for responding to commands on the channel from which the
5117 command is received.

5118 The connection established by this method persists only until the HTTP Response is
5119 transmitted or is lost due to Reader, Host or network failure.

5120 **10.4.2.9 Notification Channel**

5121 When a Reader wants to establish an HTTP connection with a Host for a Notification
5122 channel, it initiates an HTTP connection to the appropriate URL. After successful
5123 connection, the Reader sends an HTTP Request.

5124 When a Host receives an HTTP request, it SHALL accept or reject the request by sending
5125 an HTTP Response with the appropriate structure (headers, body, etc.) as defined by this
5126 transport.

5127 When a Host accepts an HTTP request on the Notification Channel, the Host MAY refuse
5128 subsequent requests for connection at that port as long as the HTTP request is being
5129 processed. If a Host chooses to accept more than a single concurrent notification request,
5130 the Host is responsible for responding to notifications on the channel from which the
5131 notification is received.

5132 The connection established by this method persists only until the HTTP Response is
5133 transmitted or is lost due to Reader, Host or network failure.

5134 **10.4.3 Connection Termination (Disconnect)**

5135 HTTP is a stateless protocol, thus the connection is terminated (or not) as per the HTTP
5136 1.1 standard. The Goodbye () command is supported but irrelevant and SHALL NOT
5137 be used over this transport.

5138 **10.4.4 Message Format**

5139 The message format used for sending and receiving data over HTTP is defined by the
5140 specific message encoding, as per the Content-Type and RP-Response-Content-Type
5141 headers. Each message encoding has a framing protocol appropriate for the formats and
5142 procedures of that encoding.

5143 The Content-Length header is always required, indicating the size of the body.

5144 Message content (HTTP Request/Response body) is never adorned with leading or
5145 trailing bytes for framing or other reasons. Use the Content-Length header to determine
5146 the size of the body, or other headers for other (non-message) information.

5147 The message format used for sending and receiving data is defined by the specific
5148 message encoding negotiated. See section 11. [Message Formats](#) for more details.

11 Message Formats

Two message formats are specified: XML and Text (non-XML). Other message formats might be added in later versions of the Reader Protocol.

11.1 XML Message Format

The following sections specify the encoding rules to be followed for all XML messages (commands, replies, and notifications) between the host and the reader, as well as the reader and the host.

The naming convention of all element tag names and attribute names follow the EPCglobal XML Design guidelines. The names SHALL use the Lower Camel Case ('LCC') convention, i.e. the leading character in the first word SHALL be in lower case and the leading characters of each subsequent word in upper case.

This chapter is divided into the following parts:

1. The scheme how an object is identified
2. The way to map simple data types
3. The way a Read Report is formatted.
4. Command XML message format (Host-to-Reader)
3. Reply XML message format (Reader-to-Host)
4. Notification XML message format (Reader-to-Host)
5. Common data types used by command, reply and notification messages
6. Vendor extension mechanism and places where they are allowed.

Please reference XML schemas in appendix A.

11.1.1 Object Identification Scheme

To specify the object parameters or the object a command relates to, the name of the object is used. Therefore, all commands that return the name of the object become useless, since the name has to be known before it can be queried. Similarly useless are methods querying for an object reference by the name of the object. Readers MAY still implement these commands however.

The following table shows the list of irrelevant commands for the XML message format:

Object Name	Method Name
ReaderDevice	getSource()
	getDataSelector()
	getNotificationChannel()
	getTrigger()
	getTagSelector()
Source	getName()

	getReadPoint()
	getReadTrigger()
	getTagSelector()
ReadPoint	getName()
TagSelector	getName()
Trigger	getName()
NotificationChannel	getName()
	getDataSelector()
	getNotificationTrigger()
DataSelector	getName()

Table 17: Irrelevant commands for XML message format

Some commands do not need explicit object identification:

- All commands of the ReaderDevice object, since this object is a singleton without a name.
- All static commands including create() commands.

11.1.2 Data Types

The data types as they are at the beginning of chapter 6 are encoded in the XML messages according to the following table:

Data type	Mapped to XML Schema type
address	xsd:anyURI
epc	epcGlobal:EPC (which maps to xsd:string) A compliant reader SHALL uniformly represent the EPC datatype using the “raw hex” URI as defined in the EPCglobal Tag Data Standards. Example: urn:epc:raw:64.x1234567890ABCDEF The hexadecimal encoding shall be case insensitive. The “raw hex” URI format specifies the length of the EPC data field.
binary	See <i>hexstring</i> .
boolean	xsd:boolean.
hexstring	xsd:hexBinary. The following rules apply to hexadecimal encoding of binary data:

	- hexadecimal encoding is case insensitive. - A host application receiving a hexstring must have a priori information regarding the length of the binary data field. Binary data fields MUST be padded with zeros prefixing the most-significant bits of the field to create a bit field length evenly divisible by four (i.e., on a nibble boundary).
integer	xsd:int
string	xsd:string
timestamp	xsd:dateTime
triggervalue	xsd:string
type[]	Lists are specified as a <code><list></code> element, with a number of <code><value></code> elements of the given type as children. Example: <pre> <list> <value>val1</value> <value>val2</value> <value>val2</value> ... </list> </pre>
tagfieldvalue	<pre> <tagfieldvalue> <name>myValue</name> <value>val1</value> </tagfieldvalue> </pre>

Table 18: Mapping of data types in XML message format

The `xsd:` namespace prefix indicates a data type as defined by the W3C XML Schema specification [XSD2].

11.1.2.1 Enumeration Objects

Where one of the enumeration objects defined in chapter 6.10 is used as a parameter in a command, it **SHALL** be represented as as an `xsd:string` with a value as defined in the relevant tables:

Object Name	Chapter containing definition of possible values
EventType	See chapter 5.5

TriggerType	See chapter 5.3
FieldName	See chapter 5.7
PreDefinedTagFieldName	See chapter 6.10.4

Table 19: Mapping of enumeration objects for XML message format

11.1.3 Read Report encoding

A `ReadReport` is returned by the `Source.readIDs()` and `Source.rawReadIDs()` commands. It can also be a part of a notification message sent asynchronously on a notification channel.

A read report SHALL be arranged in one of the two ways:

1. A list of tags with no source information.
2. List of sub reports with one sub report per source. Each source report contains a list of tags associated with that source. It MAY also have vendor defined data specific to source.

Type 1 SHALL be used when no source info is requested in the `DataSelector`, else type 2 SHALL be used.

A read report MAY also have vendor defined data common to the whole report.

Each tag in the report has the following fields:

1. Tag ID – usually an EPC.
2. Optionally the type of the tag (e.g., Gen2, ISO xxxxx)
3. Optionally a list of tag events.
4. Optionally Vendor defined data.

Each tag event has the following fields:

1. Type of the event.
2. Optionally source trigger, which was originally responsible for triggering the read cycle that lead to this event.
3. Optionally event time – either as UTC, as absolute ticks, or both.

Which data is actually reported depends on the settings of the associated `DataSelector`.

As a graphical schema, this looks as follows:

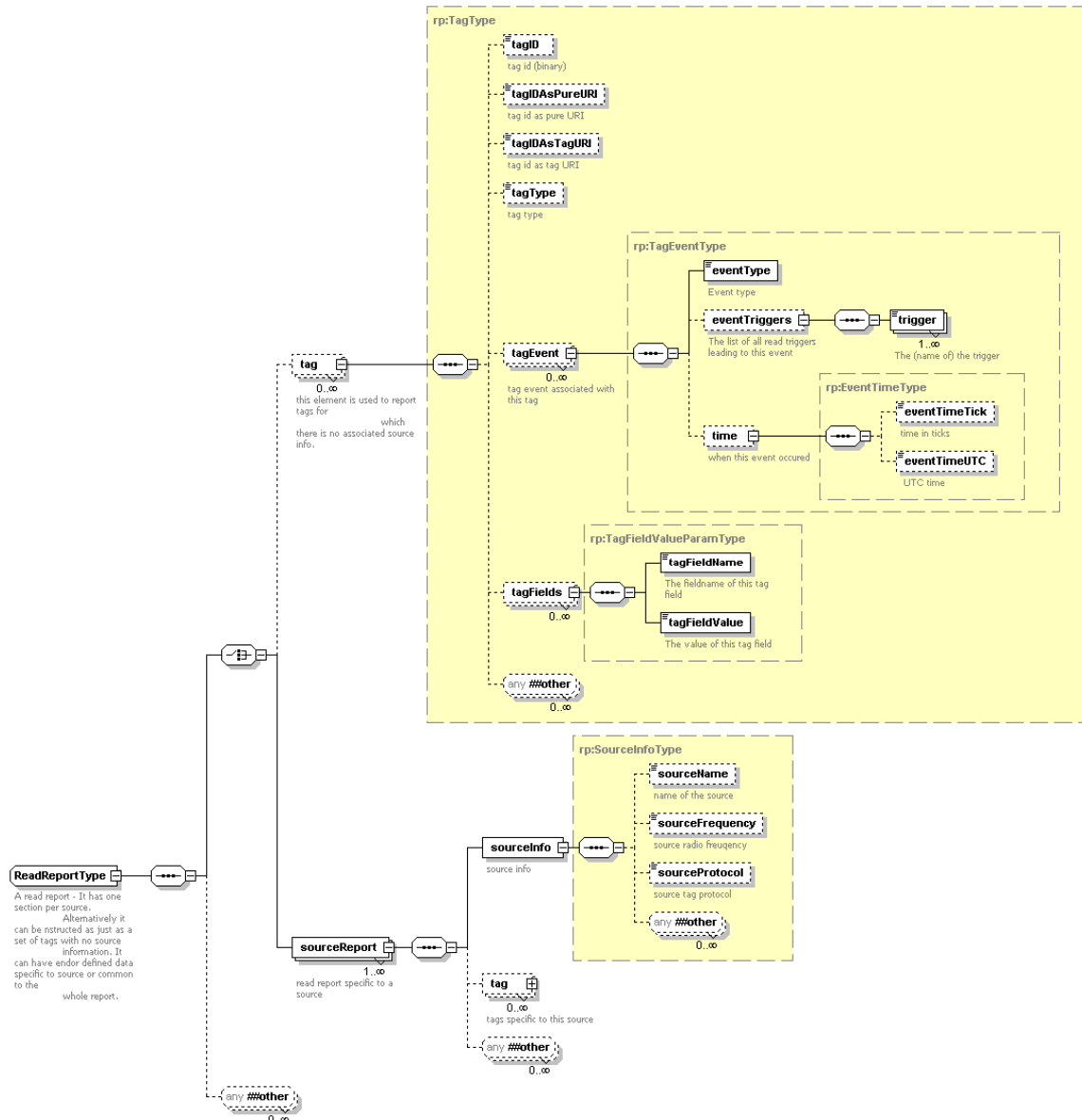


Figure 12: XML Message Format, Read Report Schema

The exact schema definition is found in appendix A.

11.1.4 Command XML message encoding (Host-To-Reader)

11.1.4.1 Message Format

Every command message is encoded within the `<command>` element. The command message body SHALL contain a message id. For non-static methods, it SHALL also contain a *target name*, identifying the object instance to which the command is applied.

5236 Commands are grouped by the object they belong to. All the commands are encoded by
5237 its straight forward conversion from UML notation. Each command is wrapped in its
5238 object type and contains its parameters as child elements.

5239 Command is encoded inside its object type as follows:

```
5240     <objectTypeName>  
5241     <commandName>parameters ...</commandName>  
5242     </objectTypeName>
```

5243 All input parameters are encoded as child elements of the command in the following
5244 manner:

```
5245     <parametername>parametervalue</parametername>
```

5246 For example, the command

```
5247     NotificationChannel.setAddress(addr: address): void
```

5248 invoked as

```
5249     channel1.setAddress("http://myhost.com:8080/EPCApp")
```

5250 SHALL be formatted as

```
5251 <command>  
5252     <id>1234</id>  
5253     <targetName>channel1</targetName>  
5254     <notificationChannel>  
5255         <setAddress>  
5256             <addr>http://myhost.com:8080/EPCApp</addr>  
5257         </setAddress>  
5258     </notificationChannel>  
5259 </command>
```

5260 Notice the command and parameter names are encoded literally as defined in the UML
5261 representation, including case sensitivity, except for the first character.

5262 **11.1.4.2 Vendor Extensions**

5263 Vendor defined commands SHALL appear under an element in their own namespace
5264 which is different from EPC namespace, within <command> element. The following,
5265 non-normative example shows this. The vendor extension command

```
5266     Acme:String.Concatenate(Left:string, Right:string):  
5267     string
```

5268 invoked as

```
5269     Acme:myString.Concatenate("abc", "xyz")
```

5270 SHALL be formatted as

```
5271 <command>  
5272     <id>1234</id>
```

```
5273     <targetName>myString</targetName>
5274     <acme:string xmlns:acme="http://www.acme.com/rp/extension">
5275         <acme:concatenate>
5276             <acme:left>abc</acme:left>
5277             <acme:right>xyz</acme:right>
5278         </acme:concatenate>
5279     </acme:string>
5280 </command>
```

5281

5282 As a graphical schema, this looks as follows (Note that for space reasons, only for the
5283 node representing the **Trigger** object all sub-elements are shown. All other objects
5284 have a similar structure, i.e. a list of all commands with their parameters):

5285

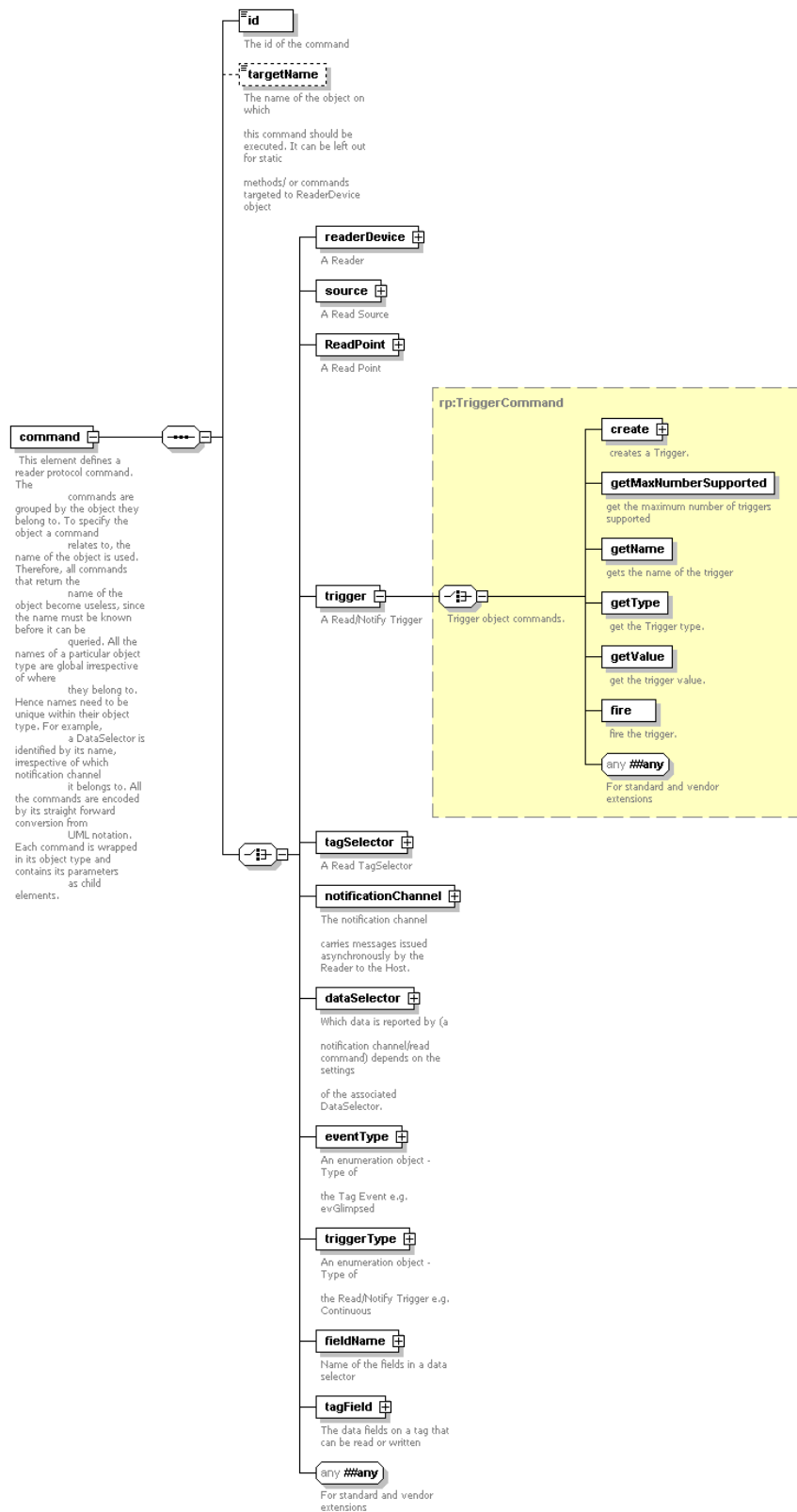


Figure 13: XML Message Format, Command Schema

5289

5290 **11.1.5 Reply XML Message Encoding (Reader-To-Host)**

5291 The Reply XML message encoding for Reader-To-Host messages uses a similar set of
5292 rules to that of the command message format, with added rules for response data and
5293 error messages.

5294 **11.1.5.1 Message Format**

5295 Every reply message is basically encoded the same as a Command message:

- 5296 1. The document element is `<reply>` instead of `<command>`
- 5297 2. The `<id>` value SHALL exactly match the value in the corresponding
5298 `<command>`
- 5299 3. The `<resultCode>` element has the error code for the command (the Error
5300 structure's "error code") with the value of NO_ERROR (see section 8.3) if the
5301 command is successful.
- 5302 4. Error information SHALL be encoded in an `<error>` node.
- 5303 5. There is no `<targetName>` element.
- 5304 6. Output is specified as child element(s) of the reply element.

5305 Simple return values(atomic data types, e.g. string) are specified as a single child element
5306 named`<returnValue>`.

5307 `<returnValue>simple return value</returnValue>`

5308 Complex return values(lists, structures, etc.) are specified as `<value>` element(s),
5309 children of `<list>`, itself a child of `<returnValue>` (aka XPath=.../retval/list/value
5310 qualifying 0+ nodes).

5311 `<returnValue>`

5312 `<list>`

5313 `<value>val1</value>`

5314 `<value>val2</value>`

5315 `<value>val2</value>`

5316 ...

5317 `</list>`

5318 `</returnValue>`

5319

5320 `<error>` element has the following fields (as child elements):

- 5321 1. Error name.
- 5322 2. Cause of the error.

5323 3. error description

5324 4. Vendor specific data.

5325 All the four fields are optional.

5326

5327 For example, the command

5328 `NotificationChannel.getAddress(): address`

5329 invoked as

5330 `channel1.getAddress()`

5331 would have a successful reply formatted as

```
5332 <reply>
5333   <id>1234</id>
5334   <resultCode>0</resultCode>
5335   <notificationChannel>
5336     <getAddress>
5337       <returnValue>http://myhost.com:8080/EPCApp</returnValue>
5338     </getAddress>
5339   </notificationChannel>
5340 </reply>
```

5341 and a failing response formatted as

```
5342 <reply>
5343   <id>1234</id>
5344   <resultCode>FFFF</resultCode>
5345   <error>
5346     <name>ERROR_VENDOR_EXTENSION</name>
5347     <cause>blah</cause>
5348     <description>blah...</description>
5349     < acme:errorInfo xmlns:acme="http://www.acme.com/rp/extension">
5350       ...anything...
5351     </acme:errorInfo>
5352   </error>
5353 </reply>
```

5354

5355 How a list of values is returned is shown in the following example. The command

5356 `NotificationChannel.getAllSources(): Source[]`

5357 invoked as

5358 `channel1.getAllSources()`

5359 would have a successful reply formatted as

5360

```

5361 <reply>
5362   <id>1234</id>
5363   <resultCode>0</resultCode>
5364   <notificationChannel>
5365     <getAllSource>
5366       <returnValue>
5367         <list>
5368           <value>source1</value>
5369           <value>source2</value>
5370           <value>source3</value>
5371         </list>
5372       </returnValue>
5373     </getAllSources>
5374   </notificationChannel>
5375 </reply>

```

As a graphical schema, this looks as follows (Note that for space reasons, only for the node representing the `Trigger` object all sub-elements are shown. All other objects have a similar structure, i.e. a list of all commands with their parameters):

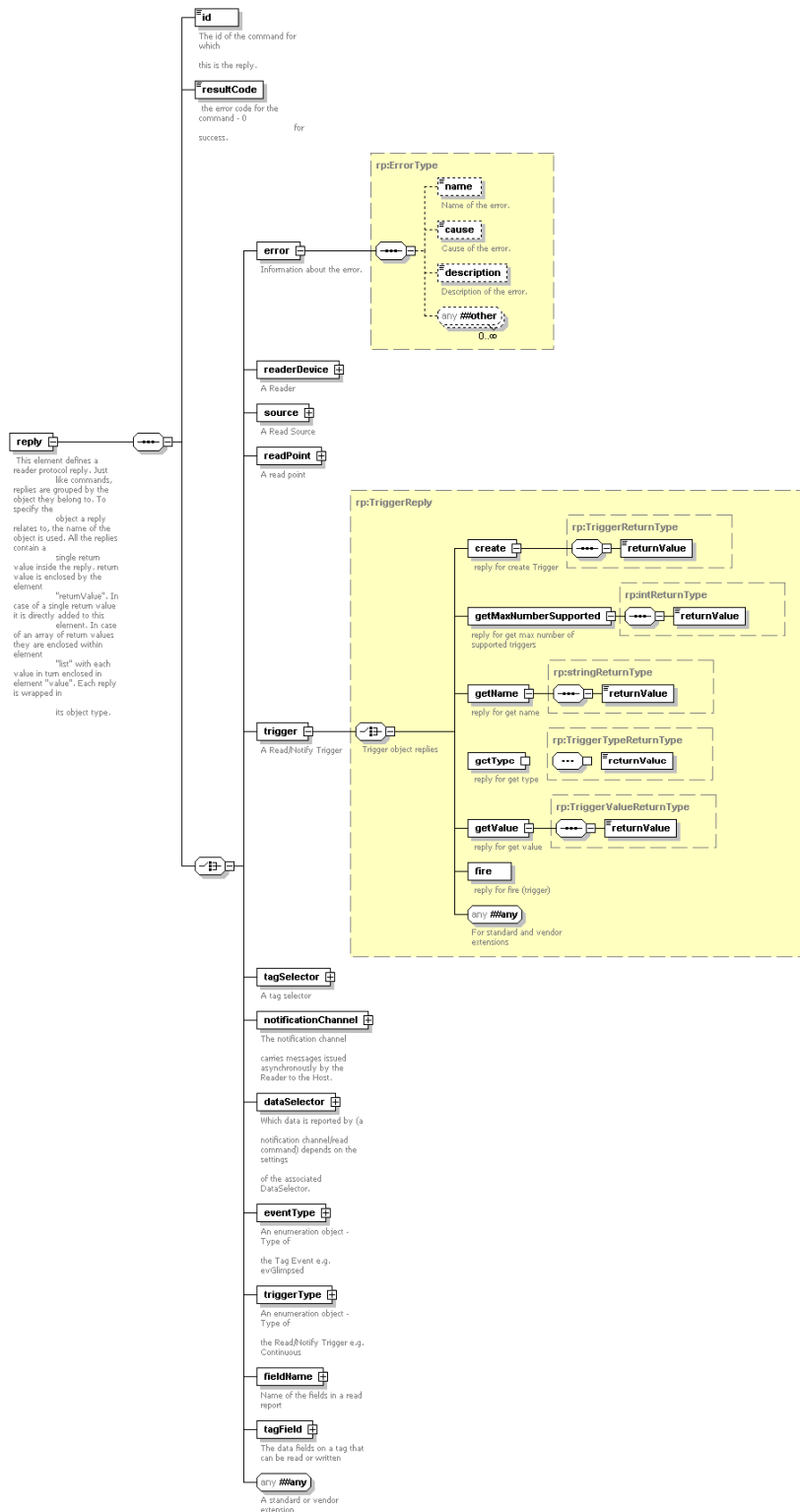


Figure 14: XML Message Format, Reply Schema

11.1.6 Notification XML message encoding (Reader-To-Host)

Every notification message is encoded within the `<notification>` element. The notification message body SHALL contain a notification id. Optionally, the following information MAY also be added:

- Information about the Reader, like its EPC or its name
- The (notification) trigger that lead to this notification
- The name of the notification channel on which this notification is sent

The content of a notification SHALL be encoded as one of the following child elements:

1. `ReadReport` – encoded as `<readReport>` element.
2. `goodbye` – encoded as `<goodBye>` element. This element has no content inside.

Which data is actually reported depends on the settings of the associated `DataSelector`.

As a graphical schema, this looks as follows (Note that the details for `ReadReport` are for space reasons not shown anymore; the exact structure of this element can be seen in the diagram in section 11.1.3):

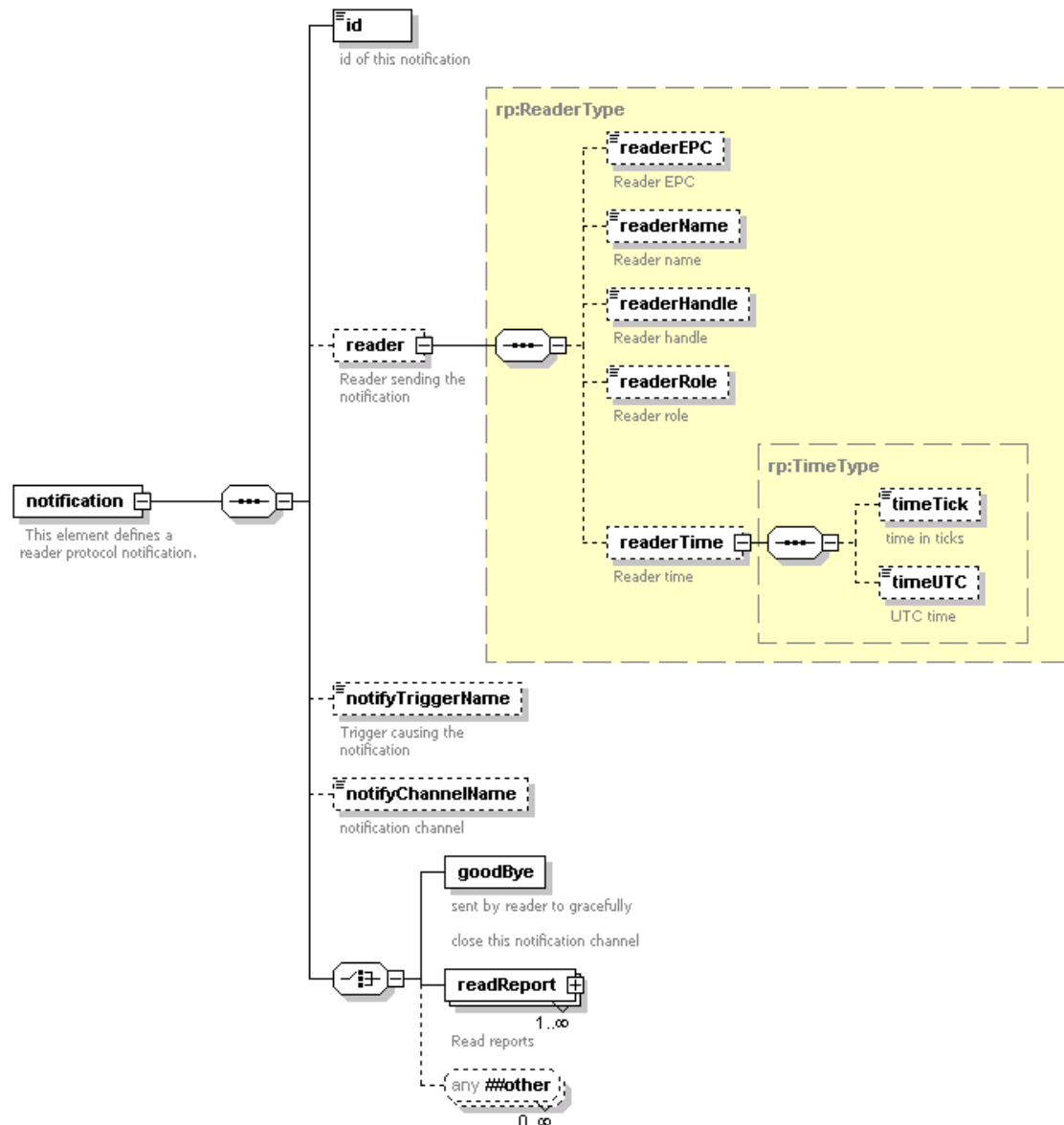


Figure 15: XML Message Format, Notification Schema

11.1.6.1 ACK/NACK

When notification ACK/NACK is enabled in a notification channel and the underlying transport doesn't support the ACKs, ACK/NACK message can be used to send ACK/NACKs by the notification receiver.

Every ACK/NACK message is encoded within the `<ackNack>` element. The notification message body SHALL contain a notification id and a boolean field `isAck` which is true if the message is an ACK.

Optionally, the following information MAY also be added in cases where id is unique across only a (Reader, notification channel) pair, to uniquely identify a notification:

- Information about the Reader, like its EPC or its name
- The name of the notification channel on which this notification is sent

As a graphical schema, ACK/NACK looks as follows :

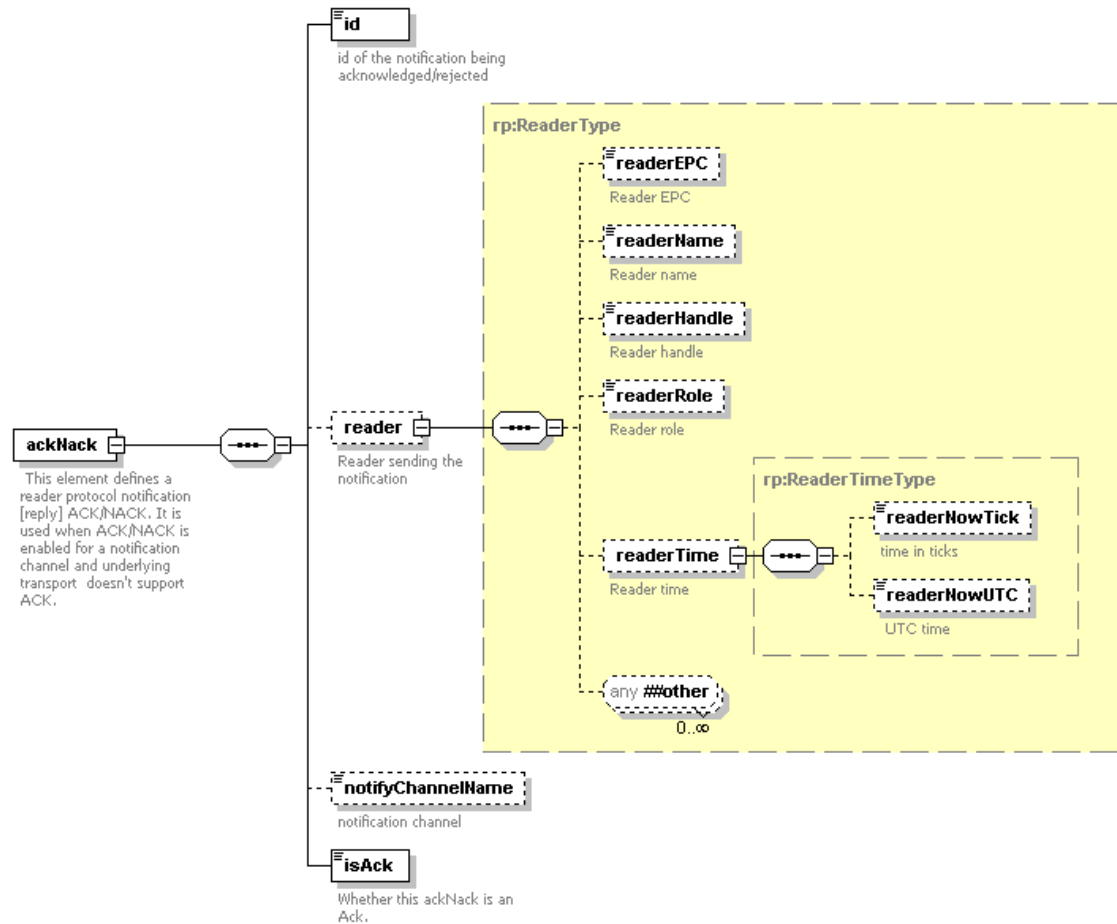


Figure 16: XML Message Format, ACK/NAK Schema

11.1.7 Vendor Extensions

Vendor extension types are allowed in specific places in the protocol. They have to be in the vendor's own namespace which has to be different from EPC namespace. These types are not validated by EPC schema. They MAY be validated by vendor specific schema.

The following table shows the list of vendor extensions allowed:

Vendor definable types
Vendor defined Command
Vendor defined Reply to vendor defined command
Vendor defined data as additional elements in Read report
Vendor defined data as additional elements in Source report
Vendor defined data as additional elements in Tag
Vendor defined data as additional elements in Notification
Vendor defined data as additional elements in Error [returned in reply]
Vendor defined attributes to any command
Vendor defined attributes to any reply
Vendor defined attributes to Read report
Vendor defined attributes to Source report
Vendor defined attributes to Tag
Vendor defined attributes to Notification
Vendor defined attributes to Error [returned in reply]

Table 20: Allowed vendor extensions of XML message format

Places where vendor defined elements are allowed are identified in the schema by,

```
<xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
```

Places where vendor defined attributes are allowed are identified in the schema by,

```
<xsd:anyAttribute processContents="lax"/>
```

11.2 Text (non-XML) Message Format

The following sections specify the encoding rules to be followed for all Text (non-XML) messages (commands, replies, and notifications) between the host and the reader, as well as the reader and the host.

Text (non-XML) syntax is defined using Augmented BNF (ABNF) as specified by IETF RFC 2234 except that “|” is used to designate alternatives and CR is defined as white space. Also, Text (non-XML) syntax is defined to be case insensitive.

The following subsections of this chapter define Text message formatting in detail.

11.2.1 External Encoding

The Text formatted message character set SHALL be in accordance with ISO 10646-1 (www.iso.org) which defines the Universal Character Set (UCS) Basic Multilingual Plane (BMP) or ISO 10646-2 (a newer addition that adds another plane to the character set). Note that ISO 10646-1:2000 (2nd edition) is officially synchronized with the Unicode 3.0 standard (www.unicode.org/book/u2.html).

The external encoding (i.e., the bits transmitted over a transport) rules for Text formatted messages SHALL adhere to the UCS Transformation Format, 8-bit form (UTF-8). This standard is defined in RFC 3629 and in Annex D of ISO 10646-1:2000.

Note there is a distinction between external encoding and the representation of a UCS character using the US ASCII printable character set. The ASCII representation of any UCS character (non-printable or otherwise) can have one of two forms:

`\uXXXX` ISO 10646-1

`\u+XXXXXX` ISO 10646-2

Where XX... is the hexadecimal code for a UCS character.

An ASCII representation of UCS characters is necessary to document UTF-8. However, note that the ASCII character sequence is not actually transmitted over the transport.

Example:

`\u0041\u0042` is encoded and transmitted as two UTF-8 characters **AB**.

It is possible to transmit the UTF-8 character sequence `\u0041\u0042`. In that case, the ASCII representation of this character sequence is `\\u0041\\u0042`.

11.2.2 Object Identification Scheme

See section 11.1.1 Object Identification Scheme. The Text (non-XML) object identification scheme follows the same rules that apply to XML formatted messages.

11.2.3 Data Types

The data types as they are at the beginning of chapter 6 are represented as literals in Text (non-XML) encoded messages according to the following table:

Data type	Mapped to
address	URI represented as a string
epc	<i>string</i> . A compliant reader SHALL uniformly represent the EPC datatype using the “raw hex” URI as defined in the EPCglobal Tag Data Standards. Example: urn:epc:raw:64.x1234567890ABCDEF The hexadecimal encoding shall be case insensitive. The “raw hex” URI format specifies the length of the EPC data field.

binary	See <i>hexstring</i> .
boolean	“1” (True) or “0” (False)
hexstring	The following rules apply to hexadecimal encoding of binary data: - hexadecimal encoding is case insensitive. - A host application receiving a hexstring must have a priori information regarding the length of the binary data field. Binary data fields MUST be padded with zeros prefixing the most-significant bits of the field to create a bit field length evenly divisible by four (i.e., on a nibble boundary).
integer	<decimal (base-10) representation>
string	Strings are non-quoted character sequences that can include backslash as an escape character. When backslash appears followed by any character (including a space character [i.e., ASCII 0x20]), the backslash is discarded and the character following it becomes part of the string text. A space following an escape character does not separate one parameter from the next. Examples: DockDoor3 Dock\ .Door\#3
timestamp	ISO 8601, i.e. YYYY-MM-DDThh:mm:ss.SSSz This format is identical to the xsd:dateTime notation used in W3C XML Schemas v1.0. See RFC 3339 for more details.
triggervalue	String
type[]	Command parameter lists are represented as comma delimited values. A command parameter that itself is a list is represented as comma delimited values (zero or more) enclosed in curly braces. A list with only one value can be represented by the value where curly braces are omitted. Two consecutive commas indicate that the element in the list has been omitted. Command reply lists are represented as either comma delimited values or multiple lines of text. A blank line indicates a null entry. Command parameter example: parm1,parm2, {parm3ValA,,parm3ValB,,parm3ValD},,parm5 Command reply example: replyValue1a,replyValue1b replyValue2

	replyValue3
	replyValue5a,,replyValue5c
tagfieldvalue	Tag field values are name-value pairs. The syntax is: tag-field-name "=" value Example: myDataField=3

Table 21: Mapping of data types in Text message format

11.2.3.1 Enumeration Objects

Where one of the enumeration objects defined in chapter 6.10 is used as a parameter in a command, it SHALL be represented as a string with a value as defined in the relevant tables:

Object Name	Chapter containing definition of possible values
EventType	See chapter 5.5
TriggerType	See chapter 5.3
FieldName	See chapter 5.7
PreDefinedTagFieldName	See chapter 6.10.4

Table 22: Mapping of enumeration objects for Text message format

11.2.4 Host Text Message Encoding (Host-To-Reader)

This section defines a Text command syntax that directly maps the Reader Protocol Object Model into text formatted messages. This is a one-to-one mapping of the Reader Protocol UML. The specific rules (non-normative) for mapping Reader Protocol UML to Text command syntax are:

1. Text message keywords are case-insensitive.
2. UML object and command names are text message keywords that MAY be recognized by a compliant reader.
3. In addition to the UML object and command name keywords, each name has an abbreviated synonym keyword defined in section 11.2.6. Abbreviated synonym names MUST be recognized by a compliant reader.
4. Every command message is encoded as a line of text.
5. The command message MAY contain a message ID as the first entry on the command line.

- 5493 6. A command message MAY contain an object type name. If the object type name
5494 is not present, then the object type MUST be implicitly defined by the command
5495 name.
- 5496 7. If an object type name is present and the object type can have multiple instances
5497 (e.g., Source), then the object type name MAY be followed by the object's
5498 instance reference name (i.e., the object target name). If the object type can have
5499 multiple instances and the object instance reference is omitted, then the object
5500 type's *default instance* MUST be assumed (e.g., Current Source or Current
5501 DataSelector).
- 5502 8. The command message MUST contain a command name.
- 5503 9. A command MAY be followed by a parameter list.
- 5504 10. If a parameter list is provided, parameters MUST be listed in the same order of
5505 occurrence as defined by the UML.

5506 A Text formatted command line is defined in terms of ABNF (see section 11.2) as
5507 follows:

5508 **command-line** = [command-id] [object-type-name ["#" target-
5509 name] "."] command-name [parameter-list] LF

5510 **command-id** = "!" dec-val

5511 **target-name** = string

5512 **parameter-list** = parameter *(", " parameter)

5513 **parameter** = primitive-val | list-val | field-val-pair

5514 **primitive-val** = bool-val | dec-val | hex-val | string

5515 **list-val** = "{" [parameter *(", " parameter)] "}"

5516 **field-val-pair** = tag-field-name "=" primitive-val

5517 **bool-val** = (TRUE | FALSE)

5518 **dec-val** = (DIGIT)+

5519 **hex-val** = (HEXDIGIT)+

5520 **string** = (STRINGCHAR1 | (ESCAPE STRINGCHAR2))+

5521 **tag-field-name** = string

5522 **DIGIT** = %x30-39

5523 **HEXDIGIT** = DIGIT | %x41-46 | %x61-67

5524 **STRINGCHAR1** = %x0022 | %x0024-002A | %x002D |

5525 %x002F-0039 | %x003B | %x003C |

5526 %x003E-005B | %x005D-007A | %x007C |

5527 %x007E-FFFF

5528 **STRINGCHAR2** = %x0020-FFFF

5529 **FALSE = %x30**

5530 **TRUE = %x31**

5531 **ESCAPE = %x5C**

5532 **SPACE = %x20**

5533 **HTAB = %x09**

5534 **WHITESPACE = SPACE | HTAB**

5535 **LF = %x0A**

5536 Examples:

5537 **Source#DockDoor5.ReadIDs myDataSelector LF**

5538 **SRC#DockDoor5.ReadIDs myDataSelector LF**

5539 **src#DockDoor5.rid myDataSelector LF**

5540 **src.rid myDataSelector LF**

5541 **rid myDataSelector LF**

5542 **rid LF**

5543 All of the above command line examples are semantically equivalent if “DockDoor5” is
5544 defined as the current Source object instance and “myDataSelector” is defined as the
5545 current DataSelector object instance.

5546 Examples including a list of objects:

5547 **ReaderDevice.removeDataSelectors{myDS1,myDS2,myDS3} LF**

5548 **rd.removeDataSelectors {myDS1 , myDS2 , myDS3} LF**

5549 **RD.remDS {myDS1,myDS2,myDS3} LF**

5550 **remDS {myDS1, myDS2, myDS3} LF**

5551 All of the above command line examples are semantically equivalent.

5552 Examples of escaped strings:

5553 **Source.addTagSelectors Read\.Filter\ \#1 LF**

5554 **SRC.rRF {\\$Filter\\$,Another\.Filter} LF**

5555 Examples with field name-value-pair:

5556 **Source.write {myField1=3,myField2=55} LF**

5557 **Source.write {myField3=100,myField4=0},{ } LF**

5558 In the first example above, the list of field name-value-pairs define the data fields and
5559 data values to be written. There is no list of TagSelector objects specified so the
5560 Source object’s list of TagSelector objects is applied. In the second example, the
5561 list of TagSelector objects parameter overrides the Source object’s list. Because the

5562 list parameter is empty, there is no tag filtering applied and the data is written to all tags
5563 in the reader's field of view.

5564 **11.2.4.1 Vendor Extensions to Host Commands**

5565 Valid vendor extensions to commands must be a command that begins with the plus
5566 character "+" followed by vendor name followed by the colon character ":". A Text
5567 formatted vendor-extended command line is defined in terms of ABNF (see section 11.2)
5568 as follows:

5569 **command-line** = [command-id] ["+" vendor-name ":"] [object-
5570 type-name ["#" target-name] "."] command-name [parameter-
5571 list] LF

5572 Example:

5573 **+ACME:rd.setAcmeAttribute 10**

5574 **11.2.5 Reader Text Message Encoding (Reader-To-Host)**

5575 Readers send messages to a host either as a reply to a command (synchronously) or as
5576 notification messages sent asynchronously. Both command replies and notification
5577 messages can include a Read Report which is defined in the first subsection below.
5578 The subsequent two subsections define the Text message format for Command
5579 Replies and Notification Messages.

5580 **11.2.5.1 Read Report Text encoding**

5581 A Text (non-XML) Read Report is based upon the following rules:

- 5582 1. Each tag is reported as a line of text.
- 5583 2. Data fields within a tag line are delimited by whitespace.
- 5584 3. Data fields specified by the corresponding DataSelector object are reported in the
5585 same order that the fields are added to the DataSelector.
- 5586 4. Whenever an "All" data field is specified (e.g., "AllEvent", "AllReader", etc.)
5587 then the corresponding data fields are reported in the same order as specified by
5588 the table enumerating data fields in section 5.7 DataSelectors.
- 5589 5. If data fields are specified in the DataSelector redundantly (e.g., both "AllEvent"
5590 and "EventType" are specified), then these data fields are reported redundantly in
5591 the Read Report and in the same order as the fields are added to the DataSelector
5592 object.

5593 A Read Report is defined in terms of ABNF (see section 11.2) as follows:

5594 **read-report** = 1*tag-line

5595 **tag-line** = data-field *("," data-field) LF

5596 There is a DataSelector object that corresponds to a Read Report and the fields listed by
5597 this DataSelector object determine what data fields will be included with each line of tag

5598 data in the Read Report. Also, the order that the data fields were added to the
5599 DataSelector object specifies the order that the fields will be listed on the tag line.

5600 Example where the Read Report's DataSelector object has just one field specified
5601 (TagID) :

5602 **F09E . . C876**

5603 **970C . . A245**

5604 **970C . . F308**

5605 Example where the Read Report's DataSelector object has multiple fields specified
5606 (TagID, EventType, ReaderNowTick, SourceName) :

5607 **F09E . . C876, evGlimpsed, 100356, mySource1**

5608 **970C . . A245, evObserved, 100407, mySource2**

5609 **970C . . F308, evLost, 100738, mySource1**

5610 **11.2.5.2 Command Reply Message Format**

5611 Command Replies are formatted as one or more lines of text. A Text Command
5612 Reply is based upon the following rules:

- 5613 1. If command-id was omitted from the command, then it should be omitted from
5614 the command reply.
- 5615 2. If present, the command-id value MUST exactly match the value in the
5616 corresponding command.
- 5617 3. Replies always include a status
- 5618 4. Reply status MUST include success/error indicator and, for errors, a status code;
5619 additional status information (e.g. cause) MAY be present.
- 5620 5. Simple return values (atomic data types, e.g. string) are specified as literal values.
- 5621 6. List of return values are reported as multiple lines of text.

5622 A Command Reply is defined in terms of ABNF (see section 11.2) as follows:

5623 **reply = [reply-header] [command-specific-reply]**

5624 **reply-header = [command-id] reply-status LF**

5625 **command-id = "!" dec-val**

5626 **reply-status = (reply-success | reply-error)**

5627 **reply-success = "OK" ["," status-code ["," status-info]]**

5628 **reply-error = "ERR" ["," status-code ["," status-info]]**

5629 **status-code = dec-val**

5630 **status-info = [result-name] ["," [result-cause]] [","**
5631 **result-description]]**

```

5632 result-name = string
5633 result-cause = string
5634 result-description = string
5635 command-specific-reply = (reply-line | read-report) LF
5636 reply-line = *(reply-value LF)
5637 reply-value = dec-val | hex-val | string | reply-list
5638 reply-list = reply-value *(", " reply-value)
5639 Examples ( command and resultant reply ):
5640     getAllSources
5641         OK
5642         mySource1
5643         mySource2
5644         mySource3
5645     !573 notificationChannel#myNC1.getDataSelector
5646         !573 OK
5647         myDataSelector5
5648     dataSelector#myDS1.getAllFieldNames
5649         OK
5650         TagID
5651         EventType
5652         ReaderNowTick
5653         SourceName
5654     source#mySource.readIDs myDS1
5655         OK
5656         F09E..C876, evGlimpsed, 100356, mySource1
5657         970C..A245, evObserved, 100407, mySource2
5658         970C..F308, evLost, 100738, mySource1
5659     !30098 invalid-command
5660         !30098 ERR, 2, ERROR_COMMAND_NOT_SUPPORTED
5661     readIDs
5662         ERR, 12, ERROR_DATASELECTOR_NOT_FOUND, "no default
5663         DataSelector specified"

```

11.2.5.3 Notification Text message encoding (Reader-To-Host)

Notification Messages are formatted as one or more lines of text. A Text Notification Message is based upon the following rules:

1. A notification message is either a *goodbye message* indicating that the reader is closing the connection or a *notification message* triggered on a particular notification channel.
2. A notification message is an optional Message Header followed by either a Read Report or a vendor specified message.
3. The Message Header MUST be present if Notification Acknowledgements (see section 10.1.2.1 Sender Handshake) have been requested.

A Notification Message is defined in terms of ABNF (see section 11.2) as follows:

```
notification = goodbye-message | notification-message
goodbye-message = "GOODBYE" LF
notification-message = ([message-header ] (read-report |
vendor-message))
message-header = message-id LF
message-id = "!" dec-val
vendor-message = "+" vendor-name ":" <vendor message
details>
```

Notification message examples (ACK/NACK requested):

```
!5034
F09E..C876,evGlimpsed,100356,mySource1
970C..A245,evObserved,100407,mySource2
970C..F308,evLost,100738,mySource1
```

```
!5035
+ACME:CUSTOM_ASYNC_EVENT
```

Notification message example (no ACK/NACK requested):

```
F09E..C876,evGlimpsed,100356,mySource1
970C..A245,evObserved,100407,mySource2
970C..F308,evLost,100738,mySource1
```

11.2.6 Abbreviations

For Text formatted commands, every UML object-type name and command name has a synonymous abbreviation. Using these abbreviations, programmers can reduce network bandwidth requirements between hosts and readers.

11.2.6.1 Object-Type Abbreviations

The following terms are synonymous in the Text format:

Name	Abbreviation
ReaderDevice	RD
Source	SRC
ReadPoint	RP
TagSelector	TS
DataSelector	DS
NotificationChannel	NC
Trigger	TRG
CommandChannel	CC
EventType	ET
TriggerType	TT
FieldName	FN
IOPorts	IOP
TagField	TF

Table 23: Object type abbreviations for Text message format

11.2.6.2 Command Abbreviations

The following tables define terms that are synonymous in the Text format.

11.2.6.2.1 ReaderDevice Command Abbreviations

Name	Abbreviation
getEPC	gE
getManufacturer	gMan
getModel	gMod

getHandle	gH
setHandle	sH
getName	gN
setName	sN
getRole	gR
setRole	sR
getTimeTicks	gTic
getTimeUTC	gUTC
setTimeUTC	sUTC
getManufacturerDescription	gMD
getCurrentSource	gCS
setCurrentSource	sCS
getCurrentDataSelector	gCDS
setCurrentDataSelector	sCDS
removeSources	rSRC
removeAllSources	raSRC
getSource	gSRC
getAllSources	gaSRC
removeDataSelectors	rDS
removeAllDataSelectors	raDS
getDataSelector	gDS
getAllDataSelectors	gaDS
removeNotificationChannels	rNC
removeAllNotificationChannels	raNC
getNotificationChannel	gNC
getAllNotificationChannels	gaNC
removeTriggers	rTRG
removeAllTriggers	raTRG
getTrigger	gTRG
getAllTriggers	gaTRG
removeTagSelectors	rTS

removeAllTagSelectors	raTS
getTagSelector	gTS
getAllTagSelectors	gaTS
removeTagFields	rTF
removeAllTagFields	raTF
getTagField	gTF
getAllTagFields	gaTF
resetToDefaultSettings	reset
reboot	boot
goodbye	bye
getReadPoint	gRP
getAllReadPoints	gaRP

Table 24: Command abbreviations of ReaderDevice object for Text message format

11.2.6.2.2 Source Command Abbreviations

Name	Abbreviation
create	c
getName	gN
isFixed	isFX
addReadPoints	aRP
removeReadPoints	rRP
removeAllReadPoints	raRP
getReadPoint	gRP
getAllReadPoints	gaRP
addReadTriggers	aRT
removeReadTriggers	rRT
removeAllReadTriggers	raRT
getReadTrigger	gRT
getAllReadTriggers	gaRT

addTagSelectors	aTS
removeTagSelectors	rTS
removeAllTagSelectors	raTS
getTagSelector	gTS
getAllTagSelectors	gaTS
getGlimpsedTimeout	gGTO
setGlimpsedTimeout	sGTO
getObservedThreshold	gOTH
setObservedThreshold	sOTH
getObservedTimeout	gOTO
setObservedTimeout	sOTO
getLostTimeout	gLTO
setLostTimeout	sLTO
rawReadIDs	rrid
readIDs	rid
read	r
writeID	wid
write	w
kill	k
getReadCyclesPerTrigger	gRCPT
setReadCyclesPerTrigger	sRCPT
getMaxReadDutyCycle	gMRDC
setMaxReadDutyCycle	sMRDC
getReadTimeout	gRTO
setReadTimeout	sRTO
getSession	gSS
setSession	sSS

Table 25: Command abbreviations of Source object for Text message format

11.2.6.2.3 ReadPoint Command Abbreviations

Name	Abbreviation
getName	gN

Table 26: Command abbreviations of ReadPoint object for Text message format

11.2.6.2.4 Trigger Command Abbreviations

Name	Abbreviation
create	c
getMaxNumberSupported	gMax
getName	gN
getType	gT
getValue	gV
fire	f

Table 27: Command abbreviations of Trigger object for Text message format

11.2.6.2.5 TagSelector Command Abbreviations

Name	Abbreviation
create	c
getMaxNumberSupported	gMx
getName	gN
getTagField	gTF
getValue	gV
getMask	gM
getInclusiveFlag	gIF

Table 28: Command abbreviations of TagSelector object for Text message format

11.2.6.2.6 NotificationChannel Command Abbreviations

Name	Abbreviation
create	c

getName	gN
getAddress	gAdr
getEffectiveAddress	gEAdr
setAddress	sAdr
getDataSelector	gDS
setDataSelector	sDS
addSources	aSRC
removeSources	rSRC
removeAllSources	raSRC
getSource	gSRC
getAllSources	gaSRC
addNotificationTriggers	aNT
removeNotificationTriggers	rNT
removeAllNotificationTriggers	raNT
getNotificationTrigger	gNT
getAllNotificationTriggers	gaNT
readQueuedData	rqd

Table 29: Command abbreviations of NotificationChannel object for Text message format

11.2.6.2.7 DataSelector Command Abbreviations

Name	Abbreviation
create	c
getName	gN
addFieldNames	aFN
removeFieldNames	rFN
removeAllFieldNames	raFN
getAllFieldNames	gaFN
addEventFilters	aEF
removeEventFilters	rEF

removeAllEventFilters	raEF
getAllEventFilters	gaEF
addTagFieldNames	aTFN
removeTagFieldNames	rTFN
removeAllTagFieldNames	raTFN
getAllTagFieldNames	gaTFN

Table 30: Command abbreviations of DataSelector object for Text message format

11.2.6.2.8 EventType Command Abbreviations

Name	Abbreviation
getSupportedTypes	gST

Table 31: Command abbreviations of EventType object for Text message format

11.2.6.2.9 TriggerType Command Abbreviations

Name	Abbreviation
getSupportedTypes	gST

Table 32: Command abbreviations of TriggerType object for Text message format

Name	Abbreviation
getSupportedNames	gSN

11.2.6.2.10 FieldName Command Abbreviations

Table 33: Command abbreviations of FieldName object for Text message format

11.2.6.2.11 TagField Command Abbreviations

Name	Abbreviation
create	c
getName	gN
getTagFieldName	gTFN
setTagFieldName	sTFN
getMemoryBank	gMB
setMemoryBank	sMB
getOffset	gOFF
setOffset	sOFF
getLength	gLEN
setLength	sLEN

Table 34: Command abbreviations of TagField object for Text message format

12 Tag RF Protocol Bindings

To help insure Reader Protocol implementation interoperability, some aspects of the Reader Protocol are defined in terms of specific tag RF protocol features. These dependencies are described in this section for the purpose of minimizing ambiguities regarding the interpretation of the Reader Protocol specification when applied to specific RF protocols. Note that the Reader Protocol does not specify or even clarify the operation of tag RF protocols.

The following table itemizes features of the Reader Protocol specification that described in this section in further detail in terms of how they are applied to specific RF protocols.

Reader Protocol Feature	Description
TAG ID	The tag's identification data element that corresponds to the Electronic Product Code (EPC).
Session	A configuration setting that can be used to help manage readers operating concurrently within RF range of one

	another.
Data field	A tag data element other than the tag's EPC.
Passwords	Data values used to control access to the tag.

Table 35: RF-Protocol specific feature bindings

The following sub-sections define specific RF protocol bindings.

12.1 EPCglobal Class0 RF Protocol Binding

12.1.1 TAG ID

The Reader Protocol TAG ID corresponds to the sole memory contents of an EPCglobal Class0 tag.

12.1.2 Session

This Reader Protocol feature does not apply to the EPCglobal Class0 RF protocol.

12.1.3 Data Field

This Reader Protocol feature does not apply to the EPCglobal Class0 RF protocol.

12.1.4 Passwords

This Reader Protocol feature does not apply to the EPCglobal Class0 RF protocol.

12.2 EPCglobal Class1 Gen1 RF Protocol Binding

12.2.1 TAG ID

The Reader Protocol TAG ID corresponds to the sole memory contents of an EPCglobal Class1 Gen1 tag.

12.2.2 Session

This Reader Protocol feature does not apply to the EPCglobal Class1 Gen1 RF protocol.

12.2.3 Data Field

This Reader Protocol feature does not apply to the EPCglobal Class1 Gen1 RF protocol.

12.2.4 Passwords

The first password in a Reader Protocol list of passwords corresponds to the EPCglobal Class 1 Gen1 RF protocol password. Passwords apply when writing or killing tags.

When writing a tag's TAG ID, if a password is specified then the tag is locked when the

5793 TAG ID is written. Subsequently, the TAG ID cannot be re-written. Also, if the tag is
5794 locked, the tag can only be killed if the corresponding password is provided with the kill
5795 operation.

5796 **12.3 EPCglobal UHF Gen2 RF Protocol Binding**

5797 **12.3.1 TAG ID**

5798 The Reader Protocol TAG ID corresponds to the UHF Gen2 protocol's *EPC memory*
5799 *bank*. The Reader Protocol TAG ID should not be confused with the UHF Gen2
5800 protocol's *TAGID memory bank*.

5801 **12.3.2 Session**

5802 This Reader Protocol feature corresponds to the EPCglobal UHF Gen2 protocol concept
5803 of *sessions and inventoried flags*.

5804 **12.3.3 Data Field**

5805 This Reader Protocol feature corresponds to the EPCglobal UHF Gen2 protocol concept
5806 of *memory range*. Reader Protocol tag data field attributes (memory bank, offset, length)
5807 map one-to-one with Gen2 memory range parameters (memory bank, pointer, length).

5808 **12.3.4 Passwords**

5809 The first password in a Reader Protocol list of passwords corresponds to the EPCglobal
5810 UHF Gen2 RF protocol *access password*. The second password in a list of passwords
5811 corresponds to the Gen2 *kill password*. The access password applies to several
5812 operations: read, write, lock, and kill tags. The kill password applies to only the kill
5813 operation.

5814

5815 **A XML Schemas**

5816 **A.1 Common Types**

5817 Common data types are defined in the Common Schema. All types end with the suffix
5818 "Type". It defines the following kind of types:

5819 **Object References:**

- 5820 1. All the command parameters or reply return value reference types are suffixed with
5821 "ParamType".
- 5822 2. All named objects are referenced using its "name". For example,
5823 DataSelectorParamType contains a reference (its name) instead of actual data
5824 selector itself.

5825

5826 For the following Reader Protocol objects a object reference type is defined:

- 5827 1. Data Selector
- 5828 2. Filter
- 5829 3. Notification channel
- 5830 4. Read point
- 5831 5. Source
- 5832 6. Trigger

5833

5834 **Enumerations:**

5835 For all enumeration objects, a type is defined which is mapped to a simple string:

- 5836 1. Event types
- 5837 2. Field names
- 5838 3. Trigger types

5839

Utility types:

The general types as explained in section 11.1.2 have their own type definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="urn:epcglobal:rp:xsd:1" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:epcglobal="urn:epcglobal:xsd:1" xmlns:rp="urn:epcglobal:rp:xsd:1" elementFormDefault="unqualified"
attributeFormDefault="unqualified" version="1.0">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      <epcglobal:copyright>Copyright ©2003-2006 Epcglobal Inc., All Rights Reserved.</epcglobal:copyright>
      <epcglobal:disclaimer>EPCglobal Inc., its members, officers, directors, employees, or
        agents shall not be liable for any injury, loss, damages, financial or otherwise,
        arising from, related to, or caused by the use of this document. The use of said
        document shall constitute your express consent to the foregoing exculpation.</epcglobal:disclaimer>
      <epcglobal:specification>Reader Protocol (RP) version 1.1</epcglobal:specification>
    </xsd:documentation>
  </xsd:annotation>
  <!-- Reader Protocol types -->
  <!-- types are ordered in alphabetical order -->
  <xsd:annotation>
    <xsd:documentation xml:lang="en"> Reader protocol common types. All types end with the
      suffix "Type". All the command parameter or reply return value reference types are
      suffixed with "ParamType". All named objects are referenced using "name". For example,
      DataSelectorParamType contains a reference (its name) instead of actual data selector
      itself. </xsd:documentation>
  </xsd:annotation>
  <xsd:import namespace="urn:epcglobal:xsd:1" schemaLocation="./EpcGlobal.xsd"/>
  <xsd:complexType name="booleanReturnType">
    <xsd:sequence>
      <xsd:element name="returnValue" type="xsd:boolean"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="intReturnType">
    <xsd:sequence>
      <xsd:element name="returnValue" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="noParamType">
    <xsd:annotation>
      <xsd:documentation>Complex type for commands that have no parameters or no return
value</xsd:documentation>
    </xsd:annotation>
  </xsd:complexType>
  <xsd:complexType name="epcReturnType">
    <xsd:sequence>
      <xsd:element name="returnValue" type="epcglobal:EPC"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="timeStampReturnType">
    <xsd:sequence>
      <xsd:element name="returnValue" type="xsd:dateTime"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="stringReturnType">
    <xsd:sequence>
```

```

5894     <xsd:element name="returnValue" type="xsd:string"/>
5895   </xsd:sequence>
5896 </xsd:complexType>
5897 <xsd:simpleType name="AddressParamType">
5898   <xsd:annotation>
5899     <xsd:documentation xml:lang="en">The address for a channel</xsd:documentation>
5900   </xsd:annotation>
5901   <xsd:restriction base="xsd:anyURI"/>
5902 </xsd:simpleType>
5903 <xsd:complexType name="AddressReturnType">
5904   <xsd:sequence>
5905     <xsd:element name="returnValue" type="rp:AddressParamType"/>
5906   </xsd:sequence>
5907 </xsd:complexType>
5908 <xsd:simpleType name="DataSelectorParamType">
5909   <xsd:annotation>
5910     <xsd:documentation xml:lang="en">The name of a data selector</xsd:documentation>
5911   </xsd:annotation>
5912   <xsd:restriction base="xsd:string"/>
5913 </xsd:simpleType>
5914 <xsd:complexType name="DataSelectorReturnType">
5915   <xsd:sequence>
5916     <xsd:element name="returnValue" type="rp:DataSelectorParamType"/>
5917   </xsd:sequence>
5918 </xsd:complexType>
5919 <xsd:complexType name="DataSelectorListParamType">
5920   <xsd:annotation>
5921     <xsd:documentation> A list of data selectors, i.e. their names</xsd:documentation>
5922   </xsd:annotation>
5923   <xsd:sequence>
5924     <xsd:element name="list">
5925       <xsd:complexType>
5926         <xsd:sequence>
5927           <xsd:element name="value" type="rp:DataSelectorParamType" maxOccurs="unbounded"/>
5928         </xsd:sequence>
5929       </xsd:complexType>
5930     </xsd:element>
5931   </xsd:sequence>
5932 </xsd:complexType>
5933 <xsd:complexType name="DataSelectorListReturnType">
5934   <xsd:sequence>
5935     <xsd:element name="returnValue" type="rp:DataSelectorListParamType"/>
5936   </xsd:sequence>
5937 </xsd:complexType>
5938 <xsd:simpleType name="EventTypeParamType">
5939   <xsd:annotation>
5940     <xsd:documentation xml:lang="en"> event type</xsd:documentation>
5941   </xsd:annotation>
5942   <xsd:restriction base="xsd:string"/>
5943 </xsd:simpleType>
5944 <xsd:complexType name="EventTypeListParamType">
5945   <xsd:annotation>
5946     <xsd:documentation> A list of event types, i.e. their names</xsd:documentation>
5947   </xsd:annotation>
5948   <xsd:sequence>

```

```

5949     <xsd:element name="list">
5950         <xsd:complexType>
5951             <xsd:sequence>
5952                 <xsd:element name="value" type="rp:EventTypeParamType" maxOccurs="unbounded"/>
5953             </xsd:sequence>
5954         </xsd:complexType>
5955     </xsd:element>
5956 </xsd:sequence>
5957 </xsd:complexType>
5958 <xsd:complexType name="EventTypeListReturnValue">
5959     <xsd:sequence>
5960         <xsd:element name="returnValue" type="rp:EventTypeListParamType"/>
5961     </xsd:sequence>
5962 </xsd:complexType>
5963 <xsd:simpleType name="FieldNameParamType">
5964     <xsd:annotation>
5965         <xsd:documentation xml:lang="en">field name</xsd:documentation>
5966     </xsd:annotation>
5967     <xsd:restriction base="xsd:string"/>
5968 </xsd:simpleType>
5969 <xsd:complexType name="FieldNameListParamType">
5970     <xsd:annotation>
5971         <xsd:documentation> A list of field names</xsd:documentation>
5972     </xsd:annotation>
5973     <xsd:sequence>
5974         <xsd:element name="list">
5975             <xsd:complexType>
5976                 <xsd:sequence>
5977                     <xsd:element name="value" type="rp:FieldNameParamType" maxOccurs="unbounded"/>
5978                 </xsd:sequence>
5979             </xsd:complexType>
5980         </xsd:element>
5981     </xsd:sequence>
5982 </xsd:complexType>
5983 <xsd:complexType name="FieldNameListReturnType">
5984     <xsd:sequence>
5985         <xsd:element name="returnValue" type="rp:FieldNameListParamType"/>
5986     </xsd:sequence>
5987 </xsd:complexType>
5988 <xsd:simpleType name="TagSelectorParamType">
5989     <xsd:annotation>
5990         <xsd:documentation xml:lang="en"> The name of a tag selector</xsd:documentation>
5991     </xsd:annotation>
5992     <xsd:restriction base="xsd:string"/>
5993 </xsd:simpleType>
5994 <xsd:complexType name="TagSelectorReturnType">
5995     <xsd:sequence>
5996         <xsd:element name="returnValue" type="rp:TagSelectorParamType"/>
5997     </xsd:sequence>
5998 </xsd:complexType>
5999 <xsd:complexType name="TagSelectorListParamType">
6000     <xsd:annotation>
6001         <xsd:documentation> A list of tag selectors, i.e. their names</xsd:documentation>
6002     </xsd:annotation>
6003     <xsd:sequence>

```

```

6004     <xsd:element name="list">
6005         <xsd:complexType>
6006             <xsd:sequence>
6007                 <xsd:element name="value" type="rp:TagSelectorParamType" maxOccurs="unbounded"/>
6008             </xsd:sequence>
6009         </xsd:complexType>
6010     </xsd:element>
6011 </xsd:sequence>
6012 </xsd:complexType>
6013 <xsd:complexType name="TagSelectorListReturnType">
6014     <xsd:sequence>
6015         <xsd:element name="returnValue" type="rp:TagSelectorListParamType"/>
6016     </xsd:sequence>
6017 </xsd:complexType>
6018 <xsd:simpleType name="TagFieldParamType">
6019     <xsd:restriction base="xsd:string"/>
6020 </xsd:simpleType>
6021 <xsd:complexType name="TagFieldReturnType">
6022     <xsd:sequence>
6023         <xsd:element name="returnValue" type="rp:TagFieldParamType"/>
6024     </xsd:sequence>
6025 </xsd:complexType>
6026 <xsd:complexType name="TagFieldListParamType">
6027     <xsd:annotation>
6028         <xsd:documentation> A list of tag fields</xsd:documentation>
6029     </xsd:annotation>
6030     <xsd:sequence>
6031         <xsd:element name="list">
6032             <xsd:complexType>
6033                 <xsd:sequence>
6034                     <xsd:element name="value" type="rp:TagFieldParamType" maxOccurs="unbounded"/>
6035                 </xsd:sequence>
6036             </xsd:complexType>
6037         </xsd:element>
6038     </xsd:sequence>
6039 </xsd:complexType>
6040 <xsd:complexType name="TagFieldListReturnType">
6041     <xsd:sequence>
6042         <xsd:element name="returnValue" type="rp:TagFieldListParamType"/>
6043     </xsd:sequence>
6044 </xsd:complexType>
6045 <xsd:simpleType name="HexStringType">
6046     <xsd:annotation>
6047         <xsd:documentation xml:lang="en"> Hex string</xsd:documentation>
6048     </xsd:annotation>
6049     <xsd:restriction base="xsd:hexBinary"/>
6050 </xsd:simpleType>
6051 <xsd:complexType name="HexStringListType">
6052     <xsd:sequence>
6053         <xsd:element name="list">
6054             <xsd:complexType>
6055                 <xsd:sequence>
6056                     <xsd:element name="value" type="xsd:hexBinary" maxOccurs="unbounded"/>
6057                 </xsd:sequence>
6058             </xsd:complexType>

```

```

6059     </xsd:element>
6060   </xsd:sequence>
6061 </xsd:complexType>
6062 <xsd:complexType name="HexStringReturnType">
6063   <xsd:sequence>
6064     <xsd:element name="returnValue" type="rp:HexStringType"/>
6065   </xsd:sequence>
6066 </xsd:complexType>
6067 <xsd:simpleType name="NotificationChannelParamType">
6068   <xsd:annotation>
6069     <xsd:documentation xml:lang="en"> name of the notification channel</xsd:documentation>
6070   </xsd:annotation>
6071   <xsd:restriction base="xsd:string"/>
6072 </xsd:simpleType>
6073 <xsd:complexType name="NotificationChannelReturnType">
6074   <xsd:sequence>
6075     <xsd:element name="returnValue" type="rp:NotificationChannelParamType"/>
6076   </xsd:sequence>
6077 </xsd:complexType>
6078 <xsd:complexType name="NotificationChannelListParamType">
6079   <xsd:annotation>
6080     <xsd:documentation> A list of notification channels, i.e. their names</xsd:documentation>
6081   </xsd:annotation>
6082   <xsd:sequence>
6083     <xsd:element name="list">
6084       <xsd:complexType>
6085         <xsd:sequence>
6086           <xsd:element name="value" type="rp:NotificationChannelParamType" maxOccurs="unbounded"/>
6087         </xsd:sequence>
6088       </xsd:complexType>
6089     </xsd:element>
6090   </xsd:sequence>
6091 </xsd:complexType>
6092 <xsd:complexType name="NotificationChannelListReturnType">
6093   <xsd:sequence>
6094     <xsd:element name="returnValue" type="rp:NotificationChannelListParamType"/>
6095   </xsd:sequence>
6096 </xsd:complexType>
6097 <xsd:simpleType name="ReadPointParamType">
6098   <xsd:annotation>
6099     <xsd:documentation xml:lang="en"> The name of a read point</xsd:documentation>
6100   </xsd:annotation>
6101   <xsd:restriction base="xsd:string"/>
6102 </xsd:simpleType>
6103 <xsd:complexType name="ReadPointReturnType">
6104   <xsd:sequence>
6105     <xsd:element name="returnValue" type="rp:ReadPointParamType"/>
6106   </xsd:sequence>
6107 </xsd:complexType>
6108 <xsd:complexType name="ReadPointListParamType">
6109   <xsd:annotation>
6110     <xsd:documentation>A list of read points, i.e. their names</xsd:documentation>
6111   </xsd:annotation>
6112   <xsd:sequence>
6113     <xsd:element name="list">

```

```

6114     <xsd:complexType>
6115         <xsd:sequence>
6116             <xsd:element name="value" type="rp:ReadPointParamType" maxOccurs="unbounded"/>
6117         </xsd:sequence>
6118     </xsd:complexType>
6119 </xsd:element>
6120 </xsd:sequence>
6121 </xsd:complexType>
6122 <xsd:complexType name="ReadPointListReturnType">
6123     <xsd:sequence>
6124         <xsd:element name="returnValue" type="rp:ReadPointListParamType"/>
6125     </xsd:sequence>
6126 </xsd:complexType>
6127 <xsd:complexType name="ReadReportType">
6128     <xsd:annotation>
6129         <xsd:documentation xml:lang="en">A read report - It has one section per source.
6130             Alternatively it can be nstructured as just as a set of tags with no source
6131             information. It can have endor defined data specific to source or common to the
6132             whole report. </xsd:documentation>
6133     </xsd:annotation>
6134     <xsd:sequence>
6135         <xsd:choice>
6136             <xsd:element name="tag" type="rp:TagType" minOccurs="0" maxOccurs="unbounded">
6137                 <xsd:annotation>
6138                     <xsd:documentation xml:lang="en">this element is used to report tags for
6139                         which there is no associated source info.</xsd:documentation>
6140                 </xsd:annotation>
6141             </xsd:element>
6142             <xsd:element name="sourceReport" maxOccurs="unbounded">
6143                 <xsd:annotation>
6144                     <xsd:documentation xml:lang="en">read report specific to a source</xsd:documentation>
6145                 </xsd:annotation>
6146                 <xsd:complexType>
6147                     <xsd:sequence>
6148                         <xsd:element name="sourceInfo" type="rp:SourceInfoType">
6149                             <xsd:annotation>
6150                                 <xsd:documentation xml:lang="en">source info</xsd:documentation>
6151                             </xsd:annotation>
6152                         </xsd:element>
6153                         <xsd:element name="tag" type="rp:TagType" minOccurs="0" maxOccurs="unbounded">
6154                             <xsd:annotation>
6155                                 <xsd:documentation xml:lang="en">tags specific to this source</xsd:documentation>
6156                             </xsd:annotation>
6157                         </xsd:element>
6158                         <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
6159                     </xsd:sequence>
6160                 </xsd:complexType>
6161             </xsd:element>
6162         </xsd:choice>
6163         <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
6164     </xsd:sequence>
6165 </xsd:complexType>
6166 <xsd:simpleType name="SourceParamType">
6167     <xsd:annotation>
6168         <xsd:documentation xml:lang="en">The name of a source</xsd:documentation>

```

```

6169     </xsd:annotation>
6170     <xsd:restriction base="xsd:string"/>
6171 </xsd:simpleType>
6172 <xsd:complexType name="SourceReturnType">
6173     <xsd:sequence>
6174         <xsd:element name="returnValue" type="rp:SourceParamType"/>
6175     </xsd:sequence>
6176 </xsd:complexType>
6177 <xsd:complexType name="SourceListParamType">
6178     <xsd:annotation>
6179         <xsd:documentation>A list of sources, i.e. names of sources</xsd:documentation>
6180     </xsd:annotation>
6181     <xsd:sequence>
6182         <xsd:element name="list">
6183             <xsd:complexType>
6184                 <xsd:sequence>
6185                     <xsd:element name="value" type="rp:SourceParamType" maxOccurs="unbounded"/>
6186                 </xsd:sequence>
6187             </xsd:complexType>
6188         </xsd:element>
6189     </xsd:sequence>
6190 </xsd:complexType>
6191 <xsd:complexType name="SourceListReturnType">
6192     <xsd:sequence>
6193         <xsd:element name="returnValue" type="rp:SourceListParamType"/>
6194     </xsd:sequence>
6195 </xsd:complexType>
6196 <xsd:complexType name="SourceInfoType">
6197     <xsd:annotation>
6198         <xsd:documentation xml:lang="en">source information</xsd:documentation>
6199     </xsd:annotation>
6200     <xsd:sequence>
6201         <xsd:element name="sourceName" type="rp:SourceParamType" minOccurs="0">
6202             <xsd:annotation>
6203                 <xsd:documentation xml:lang="en">name of the source</xsd:documentation>
6204             </xsd:annotation>
6205         </xsd:element>
6206         <xsd:element name="sourceFrequency" type="xsd:string" minOccurs="0">
6207             <xsd:annotation>
6208                 <xsd:documentation xml:lang="en">source radio frequency</xsd:documentation>
6209             </xsd:annotation>
6210         </xsd:element>
6211         <xsd:element name="sourceProtocol" type="xsd:string" minOccurs="0">
6212             <xsd:annotation>
6213                 <xsd:documentation xml:lang="en">source tag protocol</xsd:documentation>
6214             </xsd:annotation>
6215         </xsd:element>
6216         <xsd:any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
6217     </xsd:sequence>
6218 </xsd:complexType>
6219 <xsd:complexType name="StringListParamType">
6220     <xsd:sequence>
6221         <xsd:element name="list">
6222             <xsd:complexType>
6223                 <xsd:sequence>

```

```

6224         <xsd:element name="value" type="xsd:string" maxOccurs="unbounded"/>
6225     </xsd:sequence>
6226 </xsd:complexType>
6227 </xsd:element>
6228 </xsd:sequence>
6229 </xsd:complexType>
6230 <xsd:complexType name="StringListReturnType">
6231     <xsd:sequence>
6232         <xsd:element name="returnValue" type="rp:StringListParamType"/>
6233     </xsd:sequence>
6234 </xsd:complexType>
6235 <xsd:complexType name="TagEventType">
6236     <xsd:annotation>
6237         <xsd:documentation xml:lang="en">Tag Event</xsd:documentation>
6238     </xsd:annotation>
6239     <xsd:sequence>
6240         <xsd:element name="eventType" type="rp:EventTypeParamType">
6241             <xsd:annotation>
6242                 <xsd:documentation xml:lang="en">Event type</xsd:documentation>
6243             </xsd:annotation>
6244         </xsd:element>
6245         <xsd:element name="eventTriggers" minOccurs="0">
6246             <xsd:annotation>
6247                 <xsd:documentation xml:lang="en">The list of all read triggers leading to this event</xsd:documentation>
6248             </xsd:annotation>
6249             <xsd:complexType>
6250                 <xsd:sequence>
6251                     <xsd:element name="trigger" type="rp:TriggerParamType" maxOccurs="unbounded">
6252                         <xsd:annotation>
6253                             <xsd:documentation>The (name of) the trigger</xsd:documentation>
6254                         </xsd:annotation>
6255                     </xsd:element>
6256                 </xsd:sequence>
6257             </xsd:complexType>
6258         </xsd:element>
6259         <xsd:element name="time" type="rp:EventTimeType" minOccurs="0">
6260             <xsd:annotation>
6261                 <xsd:documentation xml:lang="en">when this event occurred</xsd:documentation>
6262             </xsd:annotation>
6263         </xsd:element>
6264     </xsd:sequence>
6265 </xsd:complexType>
6266 <xsd:complexType name="TagFieldValueParamType">
6267     <xsd:annotation>
6268         <xsd:documentation>Tag Field name and value pair</xsd:documentation>
6269     </xsd:annotation>
6270     <xsd:sequence>
6271         <xsd:element name="tagFieldName" type="xsd:string">
6272             <xsd:annotation>
6273                 <xsd:documentation>The fieldname of this tag field</xsd:documentation>
6274             </xsd:annotation>
6275         </xsd:element>
6276         <xsd:element name="tagFieldValue" type="xsd:hexBinary">
6277             <xsd:annotation>
6278                 <xsd:documentation>The value of this tag field</xsd:documentation>

```

```

6279     </xsd:annotation>
6280   </xsd:element>
6281 </xsd:sequence>
6282 </xsd:complexType>
6283 <xsd:complexType name="TagFieldValueListParamType">
6284   <xsd:sequence>
6285     <xsd:element name="list">
6286       <xsd:complexType>
6287         <xsd:sequence>
6288           <xsd:element name="value" type="rp:TagFieldValueParamType" maxOccurs="unbounded"/>
6289         </xsd:sequence>
6290       </xsd:complexType>
6291     </xsd:element>
6292   </xsd:sequence>
6293 </xsd:complexType>
6294 <xsd:complexType name="TagType">
6295   <xsd:annotation>
6296     <xsd:documentation xml:lang="en">tag with associated events</xsd:documentation>
6297   </xsd:annotation>
6298   <xsd:sequence>
6299     <xsd:element name="tagID" type="xsd:hexBinary" minOccurs="0">
6300       <xsd:annotation>
6301         <xsd:documentation xml:lang="en">tag id (binary)</xsd:documentation>
6302       </xsd:annotation>
6303     </xsd:element>
6304     <xsd:element name="tagIDAsPureURI" type="xsd:string" minOccurs="0">
6305       <xsd:annotation>
6306         <xsd:documentation>tag id as pure URI</xsd:documentation>
6307       </xsd:annotation>
6308     </xsd:element>
6309     <xsd:element name="tagIDAsTagURI" type="xsd:string" minOccurs="0">
6310       <xsd:annotation>
6311         <xsd:documentation>tag id as tag URI</xsd:documentation>
6312       </xsd:annotation>
6313     </xsd:element>
6314     <xsd:element name="tagType" type="xsd:string" minOccurs="0">
6315       <xsd:annotation>
6316         <xsd:documentation xml:lang="en">tag type</xsd:documentation>
6317       </xsd:annotation>
6318     </xsd:element>
6319     <xsd:element name="tagEvent" type="rp:TagEventType" minOccurs="0" maxOccurs="unbounded">
6320       <xsd:annotation>
6321         <xsd:documentation xml:lang="en">tag event associated with this tag</xsd:documentation>
6322       </xsd:annotation>
6323     </xsd:element>
6324     <xsd:element name="tagFields" type="rp:TagFieldValueParamType" minOccurs="0"
6325 maxOccurs="unbounded"/>
6326     <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
6327   </xsd:sequence>
6328 </xsd:complexType>
6329 <xsd:complexType name="EventTimeType">
6330   <xsd:annotation>
6331     <xsd:documentation xml:lang="en">Time format</xsd:documentation>
6332   </xsd:annotation>
6333   <xsd:sequence>

```

```

6334     <xsd:element name="eventTimeTick" type="xsd:string" minOccurs="0">
6335         <xsd:annotation>
6336             <xsd:documentation xml:lang="en">time in ticks</xsd:documentation>
6337         </xsd:annotation>
6338     </xsd:element>
6339     <xsd:element name="eventTimeUTC" type="xsd:dateTime" minOccurs="0">
6340         <xsd:annotation>
6341             <xsd:documentation xml:lang="en"> UTC time</xsd:documentation>
6342         </xsd:annotation>
6343     </xsd:element>
6344 </xsd:sequence>
6345 </xsd:complexType>
6346 <xsd:complexType name="ReaderTimeType">
6347     <xsd:annotation>
6348         <xsd:documentation xml:lang="en">Time format</xsd:documentation>
6349     </xsd:annotation>
6350     <xsd:sequence>
6351         <xsd:element name="readerNowTick" type="xsd:string" minOccurs="0">
6352             <xsd:annotation>
6353                 <xsd:documentation xml:lang="en">time in ticks</xsd:documentation>
6354             </xsd:annotation>
6355         </xsd:element>
6356         <xsd:element name="readerNowUTC" type="xsd:dateTime" minOccurs="0">
6357             <xsd:annotation>
6358                 <xsd:documentation xml:lang="en"> UTC time</xsd:documentation>
6359             </xsd:annotation>
6360         </xsd:element>
6361     </xsd:sequence>
6362 </xsd:complexType>
6363 <xsd:simpleType name="TriggerParamType">
6364     <xsd:annotation>
6365         <xsd:documentation xml:lang="en">The name of a trigger</xsd:documentation>
6366     </xsd:annotation>
6367     <xsd:restriction base="xsd:string"/>
6368 </xsd:simpleType>
6369 <xsd:complexType name="TriggerReturnType">
6370     <xsd:sequence>
6371         <xsd:element name="returnValue" type="rp:TriggerParamType"/>
6372     </xsd:sequence>
6373 </xsd:complexType>
6374 <xsd:complexType name="TriggerListParamType">
6375     <xsd:annotation>
6376         <xsd:documentation> A list of triggers, i.e. their names</xsd:documentation>
6377     </xsd:annotation>
6378     <xsd:sequence>
6379         <xsd:element name="list">
6380             <xsd:complexType>
6381                 <xsd:sequence>
6382                     <xsd:element name="value" type="rp:TriggerParamType" maxOccurs="unbounded"/>
6383                 </xsd:sequence>
6384             </xsd:complexType>
6385         </xsd:element>
6386     </xsd:sequence>
6387 </xsd:complexType>
6388 <xsd:complexType name="TriggerListReturnType">

```

```

6389     <xsd:sequence>
6390         <xsd:element name="returnValue" type="rp:TriggerListParamType"/>
6391     </xsd:sequence>
6392 </xsd:complexType>
6393 <xsd:simpleType name="TriggerTypeParamType">
6394     <xsd:annotation>
6395         <xsd:documentation xml:lang="en"> Trigger type</xsd:documentation>
6396     </xsd:annotation>
6397     <xsd:restriction base="xsd:string"/>
6398 </xsd:simpleType>
6399 <xsd:complexType name="TriggerTypeReturntype">
6400     <xsd:sequence>
6401         <xsd:element name="returnValue" type="rp:TriggerTypeParamType"/>
6402     </xsd:sequence>
6403 </xsd:complexType>
6404 <xsd:complexType name="TriggerTypeListReturntype">
6405     <xsd:sequence>
6406         <xsd:element name="returnValue">
6407             <xsd:complexType>
6408                 <xsd:sequence>
6409                     <xsd:element name="list">
6410                         <xsd:complexType>
6411                             <xsd:sequence>
6412                                 <xsd:element name="value" type="rp:TriggerTypeParamType" maxOccurs="unbounded"/>
6413                             </xsd:sequence>
6414                         </xsd:complexType>
6415                     </xsd:element>
6416                 </xsd:sequence>
6417             </xsd:complexType>
6418         </xsd:element>
6419     </xsd:sequence>
6420 </xsd:complexType>
6421 <xsd:simpleType name="TriggerValueParamType">
6422     <xsd:annotation>
6423         <xsd:documentation xml:lang="en"> Trigger value</xsd:documentation>
6424     </xsd:annotation>
6425     <xsd:restriction base="xsd:string"/>
6426 </xsd:simpleType>
6427 <xsd:complexType name="TriggerValueReturntype">
6428     <xsd:sequence>
6429         <xsd:element name="returnValue" type="rp:TriggerValueParamType"/>
6430     </xsd:sequence>
6431 </xsd:complexType>
6432 </xsd:schema>

```

A.2 Command Schema

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="urn:epcglobal:rp:xsd:1" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:epcglobal="urn:epcglobal:xsd:1" xmlns:rp="urn:epcglobal:rp:xsd:1" elementFormDefault="unqualified"
attributeFormDefault="unqualified" version="1.0">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      <epcglobal:copyright>Copyright ©2003-2006 Epcglobal Inc., All Rights Reserved.</epcglobal:copyright>
      <epcglobal:disclaimer>EPCglobal Inc., its members, officers, directors, employees, or
        agents shall not be liable for any injury, loss, damages, financial or otherwise,
        arising from, related to, or caused by the use of this document. The use of said
        document shall constitute your express consent to the foregoing exculpation.</epcglobal:disclaimer>
      <epcglobal:specification>Reader Protocol (RP) version 1.1</epcglobal:specification>
    </xsd:documentation>
  </xsd:annotation>
  <xsd:include schemaLocation="RpCommon.xsd"/>
  <xsd:import namespace="urn:epcglobal:xsd:1" schemaLocation="./EpcGlobal.xsd"/>
  <!-- Reader Protocol Command -->
  <xsd:element name="command">
    <xsd:annotation>
      <xsd:documentation xml:lang="en"> This element defines a reader protocol command. The
        commands are grouped by the object they belong to. To specify the object a command
        relates to, the name of the object is used. Therefore, all commands that return the
        name of the object become useless, since the name must be known before it can be
        queried. All the names of a particular object type are global irrespective of where
        they belong to. Hence names need to be unique within their object type. For example,
        a DataSelector is identified by its name, irrespective of which notification channel
        it belongs to. All the commands are encoded by its straight forward conversion from
        UML notation. Each command is wrapped in its object type and contains its parameters
        as child elements. </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="id" type="xsd:string">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">The id of the command</xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="targetName" type="xsd:string" minOccurs="0">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">The name of the object on which
              this command should be executed. It can be left out for static
              methods/ or commands targeted to ReaderDevice object</xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:choice>
          <xsd:element name="readerDevice" type="rp:ReaderDeviceCommand">
            <xsd:annotation>
              <xsd:documentation xml:lang="en">A Reader</xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="source" type="rp:SourceCommand">
            <xsd:annotation>
              <xsd:documentation xml:lang="en">A Read Source</xsd:documentation>
            </xsd:annotation>
          </xsd:element>
        </xsd:choice>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

```

```

6487     </xsd:annotation>
6488 </xsd:element>
6489 <xsd:element name="ReadPoint" type="rp:ReadPointCommand">
6490     <xsd:annotation>
6491         <xsd:documentation>A Read Point</xsd:documentation>
6492     </xsd:annotation>
6493 </xsd:element>
6494 <xsd:element name="trigger" type="rp:TriggerCommand">
6495     <xsd:annotation>
6496         <xsd:documentation xml:lang="en">A Read/Notify Trigger</xsd:documentation>
6497     </xsd:annotation>
6498 </xsd:element>
6499 <xsd:element name="tagSelector" type="rp:TagSelectorCommand">
6500     <xsd:annotation>
6501         <xsd:documentation xml:lang="en">A Read TagSelector</xsd:documentation>
6502     </xsd:annotation>
6503 </xsd:element>
6504 <xsd:element name="notificationChannel" type="rp:NotificationChannelCommand">
6505     <xsd:annotation>
6506         <xsd:documentation xml:lang="en">The notification channel
6507             carries messages issued asynchronously by the Reader to the Host.</xsd:documentation>
6508     </xsd:annotation>
6509 </xsd:element>
6510 <xsd:element name="dataSelector" type="rp:DataSelectorCommand">
6511     <xsd:annotation>
6512         <xsd:documentation xml:lang="en">Which data is reported by (a
6513             notification channel/read command) depends on the settings
6514             of the associated DataSelector.</xsd:documentation>
6515     </xsd:annotation>
6516 </xsd:element>
6517 <xsd:element name="eventType" type="rp:EventTypeCommand">
6518     <xsd:annotation>
6519         <xsd:documentation xml:lang="en">An enumeration object - Type of
6520             the Tag Event e.g. evGlimpsed</xsd:documentation>
6521     </xsd:annotation>
6522 </xsd:element>
6523 <xsd:element name="triggerType" type="rp:TriggerTypeCommand">
6524     <xsd:annotation>
6525         <xsd:documentation xml:lang="en">An enumeration object - Type of
6526             the Read/Notify Trigger e.g. Continuous</xsd:documentation>
6527     </xsd:annotation>
6528 </xsd:element>
6529 <xsd:element name="fieldName" type="rp:FieldNameCommand">
6530     <xsd:annotation>
6531         <xsd:documentation xml:lang="en">Name of the fields in a data selector</xsd:documentation>
6532     </xsd:annotation>
6533 </xsd:element>
6534 <xsd:element name="tagField" type="rp:TagFieldCommand">
6535     <xsd:annotation>
6536         <xsd:documentation>The data fields on a tag that can be read or written</xsd:documentation>
6537     </xsd:annotation>
6538 </xsd:element>
6539 <xsd:any namespace="##any" processContents="lax">
6540     <xsd:annotation>
6541         <xsd:documentation>For standard and vendor extensions</xsd:documentation>

```

```

6542         </xsd:annotation>
6543     </xsd:any>
6544     <!-- Reader Protocol (Target) Object Types -->
6545     <!-- elements inside are ordered in alphabetical order -->
6546 </xsd:choice>
6547 </xsd:sequence>
6548 </xsd:complexType>
6549 </xsd:element>
6550 <!-- Reader Protocol (Target) Object Types -->
6551 <!-- types are ordered in alphabetical order -->
6552 <!-- inside each type commands are listed in the order in which it is defined in the protocol specification. -->
6553 <xsd:complexType name="DataSelectorCommand">
6554     <xsd:choice>
6555         <xsd:annotation>
6556             <xsd:documentation xml:lang="en"> Data Selector object commands </xsd:documentation>
6557         </xsd:annotation>
6558         <xsd:element name="create">
6559             <xsd:annotation>
6560                 <xsd:documentation xml:lang="en"> static method to create a data selector. </xsd:documentation>
6561             </xsd:annotation>
6562             <xsd:complexType>
6563                 <xsd:sequence>
6564                     <xsd:element name="name" type="xsd:string">
6565                         <xsd:annotation>
6566                             <xsd:documentation xml:lang="en"> Data Selector name </xsd:documentation>
6567                         </xsd:annotation>
6568                     </xsd:element>
6569                 </xsd:sequence>
6570             </xsd:complexType>
6571         </xsd:element>
6572         <xsd:element name="getName" type="rp:noParamType">
6573             <xsd:annotation>
6574                 <xsd:documentation>get the data selector name</xsd:documentation>
6575             </xsd:annotation>
6576         </xsd:element>
6577         <xsd:element name="addFieldNames">
6578             <xsd:annotation>
6579                 <xsd:documentation xml:lang="en"> Add fields to a data selector. </xsd:documentation>
6580             </xsd:annotation>
6581             <xsd:complexType>
6582                 <xsd:sequence>
6583                     <xsd:element name="fieldNames" type="rp:FieldNameListParamType">
6584                         <xsd:annotation>
6585                             <xsd:documentation xml:lang="en"> fields to add. </xsd:documentation>
6586                         </xsd:annotation>
6587                     </xsd:element>
6588                 </xsd:sequence>
6589             </xsd:complexType>
6590         </xsd:element>
6591         <xsd:element name="removeFieldNames">
6592             <xsd:annotation>
6593                 <xsd:documentation xml:lang="en"> remove fields from a data selector. </xsd:documentation>
6594             </xsd:annotation>
6595         </xsd:complexType>
6596     </xsd:sequence>

```

```

6597     <xsd:element name="fieldNames" type="rp:FieldNameListParamType">
6598         <xsd:annotation>
6599             <xsd:documentation xml:lang="en"> fields to remove. </xsd:documentation>
6600         </xsd:annotation>
6601     </xsd:element>
6602 </xsd:sequence>
6603 </xsd:complexType>
6604 </xsd:element>
6605 <xsd:element name="removeAllFieldNames" type="rp:noParamType">
6606     <xsd:annotation>
6607         <xsd:documentation xml:lang="en"> remove all fields from a data selector. </xsd:documentation>
6608     </xsd:annotation>
6609 </xsd:element>
6610 <xsd:element name="getAllFieldNames" type="rp:noParamType">
6611     <xsd:annotation>
6612         <xsd:documentation xml:lang="en"> get all fields in a data selector. </xsd:documentation>
6613     </xsd:annotation>
6614 </xsd:element>
6615 <xsd:element name="addEventFilters">
6616     <xsd:annotation>
6617         <xsd:documentation xml:lang="en"> add eventTypes to be filtered to a data
6618             selector. </xsd:documentation>
6619     </xsd:annotation>
6620     <xsd:complexType>
6621         <xsd:sequence>
6622             <xsd:element name="eventType" type="rp:EventTypeListParamType">
6623                 <xsd:annotation>
6624                     <xsd:documentation xml:lang="en"> eventTypes to filter. </xsd:documentation>
6625                 </xsd:annotation>
6626             </xsd:element>
6627         </xsd:sequence>
6628     </xsd:complexType>
6629 </xsd:element>
6630 <xsd:element name="removeEventFilters">
6631     <xsd:annotation>
6632         <xsd:documentation xml:lang="en"> remove eventTypes to be filtered from a data
6633             selector. </xsd:documentation>
6634     </xsd:annotation>
6635     <xsd:complexType>
6636         <xsd:sequence>
6637             <xsd:element name="eventType" type="rp:EventTypeListParamType">
6638                 <xsd:annotation>
6639                     <xsd:documentation xml:lang="en"> eventTypes to remove from filtered
6640                         eventTypes. </xsd:documentation>
6641                 </xsd:annotation>
6642             </xsd:element>
6643         </xsd:sequence>
6644     </xsd:complexType>
6645 </xsd:element>
6646 <xsd:element name="removeAllEventFilters" type="rp:noParamType">
6647     <xsd:annotation>
6648         <xsd:documentation xml:lang="en"> set a data selector to filter no eventTypes. </xsd:documentation>
6649     </xsd:annotation>
6650 </xsd:element>
6651 <xsd:element name="getAllEventFilters" type="rp:noParamType">

```

```

6652     <xsd:annotation>
6653         <xsd:documentation xml:lang="en"> get all filtered eventTypes in a data
6654             selector. </xsd:documentation>
6655     </xsd:annotation>
6656 </xsd:element>
6657 <xsd:element name="addTagFieldNames">
6658     <xsd:annotation>
6659         <xsd:documentation>add field names of TagFields that should be read </xsd:documentation>
6660     </xsd:annotation>
6661     <xsd:complexType>
6662         <xsd:sequence>
6663             <xsd:element name="fieldNames" type="rp:StringListParamType"/>
6664         </xsd:sequence>
6665     </xsd:complexType>
6666 </xsd:element>
6667 <xsd:element name="removeTagFieldNames">
6668     <xsd:annotation>
6669         <xsd:documentation>remove field names of TagFields that should be read</xsd:documentation>
6670     </xsd:annotation>
6671     <xsd:complexType>
6672         <xsd:sequence>
6673             <xsd:element name="fieldNames" type="rp:StringListParamType"/>
6674         </xsd:sequence>
6675     </xsd:complexType>
6676 </xsd:element>
6677 <xsd:element name="removeAllTagFieldNames" type="rp:noParamType">
6678     <xsd:annotation>
6679         <xsd:documentation>remove all field names of TagFields that should be read</xsd:documentation>
6680     </xsd:annotation>
6681 </xsd:element>
6682 <xsd:element name="getAllTagFieldNames" type="rp:noParamType">
6683     <xsd:annotation>
6684         <xsd:documentation>get all field names of TagFields that should be read</xsd:documentation>
6685     </xsd:annotation>
6686 </xsd:element>
6687 <xsd:any namespace="##any" processContents="lax">
6688     <xsd:annotation>
6689         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
6690     </xsd:annotation>
6691 </xsd:any>
6692 </xsd:choice>
6693 </xsd:complexType>
6694 <xsd:complexType name="EventTypeCommand">
6695     <xsd:annotation>
6696         <xsd:documentation xml:lang="en"> Event type enumeration object commands. </xsd:documentation>
6697     </xsd:annotation>
6698     <xsd:choice>
6699         <xsd:element name="getSupportedTypes" type="rp:noParamType">
6700             <xsd:annotation>
6701                 <xsd:documentation xml:lang="en"> get all the eventTypes supported in the reader
6702                     device. </xsd:documentation>
6703             </xsd:annotation>
6704         </xsd:element>
6705     </xsd:choice>
6706 </xsd:complexType>

```

```

6707 <xsd:complexType name="FieldNameCommand">
6708   <xsd:annotation>
6709     <xsd:documentation xml:lang="en"> Field Name enumeration object commands. </xsd:documentation>
6710   </xsd:annotation>
6711   <xsd:choice>
6712     <xsd:element name="getSupportedNames" type="rp:noParamType">
6713       <xsd:annotation>
6714         <xsd:documentation xml:lang="en"> get all the field names of a data selector
6715           supported in the reader device. </xsd:documentation>
6716       </xsd:annotation>
6717     </xsd:element>
6718   </xsd:choice>
6719 </xsd:complexType>
6720 <xsd:complexType name="NotificationChannelCommand">
6721   <xsd:annotation>
6722     <xsd:documentation xml:lang="en"> Notification channel object commands </xsd:documentation>
6723   </xsd:annotation>
6724   <xsd:choice>
6725     <xsd:element name="create">
6726       <xsd:annotation>
6727         <xsd:documentation xml:lang="en"> static method to create a notification
6728           channel. </xsd:documentation>
6729       </xsd:annotation>
6730       <xsd:complexType>
6731         <xsd:sequence>
6732           <xsd:element name="name" type="xsd:string">
6733             <xsd:annotation>
6734               <xsd:documentation xml:lang="en"> notification channel name. </xsd:documentation>
6735             </xsd:annotation>
6736           </xsd:element>
6737           <xsd:element name="address" type="rp:AddressParamType">
6738             <xsd:annotation>
6739               <xsd:documentation xml:lang="en"> notification channel address. </xsd:documentation>
6740             </xsd:annotation>
6741           </xsd:element>
6742         </xsd:sequence>
6743       </xsd:complexType>
6744     </xsd:element>
6745     <xsd:element name="getName" type="rp:noParamType">
6746       <xsd:annotation>
6747         <xsd:documentation>get the name of the notification channel</xsd:documentation>
6748       </xsd:annotation>
6749     </xsd:element>
6750     <xsd:element name="getAddress" type="rp:noParamType">
6751       <xsd:annotation>
6752         <xsd:documentation xml:lang="en"> get the address of a notification channel. </xsd:documentation>
6753       </xsd:annotation>
6754     </xsd:element>
6755     <xsd:element name="getEffectiveAddress" type="rp:noParamType">
6756       <xsd:annotation>
6757         <xsd:documentation>get the effective address of the notification channel</xsd:documentation>
6758       </xsd:annotation>
6759     </xsd:element>
6760     <xsd:element name="setAddress">
6761       <xsd:annotation>

```

```

6762      <xsd:documentation xml:lang="en"> set the address of a notification channel. </xsd:documentation>
6763    </xsd:annotation>
6764    <xsd:complexType>
6765      <xsd:sequence>
6766        <xsd:element name="address" type="rp:AddressParamType">
6767          <xsd:annotation>
6768            <xsd:documentation xml:lang="en"> address to set. </xsd:documentation>
6769          </xsd:annotation>
6770        </xsd:element>
6771      </xsd:sequence>
6772    </xsd:complexType>
6773  </xsd:element>
6774  <xsd:element name="getDataSelector" type="rp:noParamType">
6775    <xsd:annotation>
6776      <xsd:documentation xml:lang="en"> get the data selector of a notification
6777        channel. </xsd:documentation>
6778    </xsd:annotation>
6779  </xsd:element>
6780  <xsd:element name="setDataSelector">
6781    <xsd:annotation>
6782      <xsd:documentation xml:lang="en"> set the data selector of a notification
6783        channel. </xsd:documentation>
6784    </xsd:annotation>
6785    <xsd:complexType>
6786      <xsd:sequence>
6787        <xsd:element name="dataSelector" type="rp:DataSelectorParamType">
6788          <xsd:annotation>
6789            <xsd:documentation xml:lang="en"> data selector to set. </xsd:documentation>
6790          </xsd:annotation>
6791        </xsd:element>
6792      </xsd:sequence>
6793    </xsd:complexType>
6794  </xsd:element>
6795  <xsd:element name="addSources">
6796    <xsd:annotation>
6797      <xsd:documentation xml:lang="en"> add sources (whose events will be reported
6798        through this channel) to the notification channel. </xsd:documentation>
6799    </xsd:annotation>
6800    <xsd:complexType>
6801      <xsd:sequence>
6802        <xsd:element name="sources" type="rp:SourceListParamType">
6803          <xsd:annotation>
6804            <xsd:documentation xml:lang="en"> sources to add(whose events will be
6805              reported through this channel). </xsd:documentation>
6806          </xsd:annotation>
6807        </xsd:element>
6808      </xsd:sequence>
6809    </xsd:complexType>
6810  </xsd:element>
6811  <xsd:element name="removeSources">
6812    <xsd:annotation>
6813      <xsd:documentation xml:lang="en"> remove sources from the list of sources, whose
6814        events will be reported through this channel. </xsd:documentation>
6815    </xsd:annotation>
6816  </xsd:complexType>

```

```

6817     <xsd:sequence>
6818         <xsd:element name="sources" type="rp:SourceListParamType">
6819             <xsd:annotation>
6820                 <xsd:documentation xml:lang="en"> sources to remove from the list of
6821                     sources, whose events will be reported through this channel. </xsd:documentation>
6822             </xsd:annotation>
6823         </xsd:element>
6824     </xsd:sequence>
6825 </xsd:complexType>
6826 </xsd:element>
6827 <xsd:element name="removeAllSources" type="rp:noParamType">
6828     <xsd:annotation>
6829         <xsd:documentation xml:lang="en"> clear the list of sources, whose events will
6830             be reported through this channel. </xsd:documentation>
6831     </xsd:annotation>
6832 </xsd:element>
6833 <xsd:element name="getSource">
6834     <xsd:annotation>
6835         <xsd:documentation xml:lang="en"> get the named source from the list of sources,
6836             whose events will be reported through this channel. </xsd:documentation>
6837     </xsd:annotation>
6838 <xsd:complexType>
6839     <xsd:sequence>
6840         <xsd:element name="name" type="xsd:string">
6841             <xsd:annotation>
6842                 <xsd:documentation xml:lang="en"> name of the source to get. </xsd:documentation>
6843             </xsd:annotation>
6844         </xsd:element>
6845     </xsd:sequence>
6846 </xsd:complexType>
6847 </xsd:element>
6848 <xsd:element name="getAllSources" type="rp:noParamType">
6849     <xsd:annotation>
6850         <xsd:documentation xml:lang="en"> get the list of sources, whose events will be
6851             reported through this channel. </xsd:documentation>
6852     </xsd:annotation>
6853 </xsd:element>
6854 <xsd:element name="addNotificationTriggers">
6855     <xsd:annotation>
6856         <xsd:documentation xml:lang="en"> add triggers (which will cause the events to
6857             be reported to the host through this channel) to the notification channel. </xsd:documentation>
6858     </xsd:annotation>
6859 <xsd:complexType>
6860     <xsd:sequence>
6861         <xsd:element name="triggers" type="rp:TriggerListParamType">
6862             <xsd:annotation>
6863                 <xsd:documentation xml:lang="en"> triggers to add(which will cause
6864                     the events to be reported to the host through this channel) </xsd:documentation>
6865             </xsd:annotation>
6866         </xsd:element>
6867     </xsd:sequence>
6868 </xsd:complexType>
6869 </xsd:element>
6870 <xsd:element name="removeNotificationTriggers">
6871     <xsd:annotation>

```

```

6872      <xsd:documentation xml:lang="en"> remove triggers from the list of triggers,
6873      which will cause the events to be reported to the host through this channel. </xsd:documentation>
6874    </xsd:annotation>
6875    <xsd:complexType>
6876      <xsd:sequence>
6877        <xsd:element name="triggers" type="rp:TriggerListParamType">
6878          <xsd:annotation>
6879            <xsd:documentation xml:lang="en"> triggers to remove from the list of
6880            triggers, which will cause the events to be reported to the host
6881            through this channel. </xsd:documentation>
6882          </xsd:annotation>
6883        </xsd:element>
6884      </xsd:sequence>
6885    </xsd:complexType>
6886  </xsd:element>
6887  <xsd:element name="removeAllNotificationTriggers" type="rp:noParamType">
6888    <xsd:annotation>
6889      <xsd:documentation xml:lang="en"> clear the list of triggers, which will cause
6890      the events to be reported to the host through this channel. </xsd:documentation>
6891    </xsd:annotation>
6892  </xsd:element>
6893  <xsd:element name="getNotificationTrigger">
6894    <xsd:annotation>
6895      <xsd:documentation xml:lang="en"> get the named trigger from the list of
6896      triggers, which will cause the events to be reported to the host through
6897      this channel. </xsd:documentation>
6898    </xsd:annotation>
6899    <xsd:complexType>
6900      <xsd:sequence>
6901        <xsd:element name="name" type="xsd:string">
6902          <xsd:annotation>
6903            <xsd:documentation xml:lang="en"> name of the trigger to get. </xsd:documentation>
6904          </xsd:annotation>
6905        </xsd:element>
6906      </xsd:sequence>
6907    </xsd:complexType>
6908  </xsd:element>
6909  <xsd:element name="getAllNotificationTriggers" type="rp:noParamType">
6910    <xsd:annotation>
6911      <xsd:documentation xml:lang="en"> get the list of triggers, which will cause the
6912      events to be reported to the host through this channel. </xsd:documentation>
6913    </xsd:annotation>
6914  </xsd:element>
6915  <xsd:element name="readQueuedData">
6916    <xsd:annotation>
6917      <xsd:documentation xml:lang="en"> reads the event data associated with this
6918      channel through control channel synchronously. </xsd:documentation>
6919    </xsd:annotation>
6920    <xsd:complexType>
6921      <xsd:sequence>
6922        <xsd:element name="clearBuffer" type="xsd:boolean">
6923          <xsd:annotation>
6924            <xsd:documentation xml:lang="en"> flag to indicate if the buffer should be cleared or
6925            not</xsd:documentation>
6926          </xsd:annotation>

```

```

6927         </xsd:element>
6928     </xsd:sequence>
6929 </xsd:complexType>
6930 </xsd:element>
6931 <xsd:any namespace="##any" processContents="lax">
6932     <xsd:annotation>
6933         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
6934     </xsd:annotation>
6935 </xsd:any>
6936 </xsd:choice>
6937 </xsd:complexType>
6938 <xsd:complexType name="ReaderDeviceCommand">
6939     <xsd:annotation>
6940         <xsd:documentation xml:lang="en"> Reader Device object commands. </xsd:documentation>
6941     </xsd:annotation>
6942     <xsd:choice>
6943         <xsd:element name="getEPC" type="rp:noParamType">
6944             <xsd:annotation>
6945                 <xsd:documentation xml:lang="en"> get the EPC code of the Reader Device. </xsd:documentation>
6946             </xsd:annotation>
6947         </xsd:element>
6948         <xsd:element name="getManufacturer" type="rp:noParamType">
6949             <xsd:annotation>
6950                 <xsd:documentation xml:lang="en"> get the manufacturer of the Reader Device. </xsd:documentation>
6951             </xsd:annotation>
6952         </xsd:element>
6953         <xsd:element name="getModel" type="rp:noParamType">
6954             <xsd:annotation>
6955                 <xsd:documentation xml:lang="en"> get the Reader Device Model. </xsd:documentation>
6956             </xsd:annotation>
6957         </xsd:element>
6958         <xsd:element name="getHandle" type="rp:noParamType">
6959             <xsd:annotation>
6960                 <xsd:documentation xml:lang="en"> get the Handle of the Reader Device. </xsd:documentation>
6961             </xsd:annotation>
6962         </xsd:element>
6963         <xsd:element name="setHandle">
6964             <xsd:annotation>
6965                 <xsd:documentation xml:lang="en"> set the Handle of the Reader Device. </xsd:documentation>
6966             </xsd:annotation>
6967         <xsd:complexType>
6968             <xsd:sequence>
6969                 <xsd:element name="handle" type="xsd:int">
6970                     <xsd:annotation>
6971                         <xsd:documentation xml:lang="en"> Handle to set. </xsd:documentation>
6972                     </xsd:annotation>
6973                 </xsd:element>
6974             </xsd:sequence>
6975         </xsd:complexType>
6976     </xsd:element>
6977     <xsd:element name="getName" type="rp:noParamType">
6978         <xsd:annotation>
6979             <xsd:documentation>get the name of the reader</xsd:documentation>
6980         </xsd:annotation>
6981     </xsd:element>

```

```

6982 <xsd:element name="setName">
6983   <xsd:annotation>
6984     <xsd:documentation xml:lang="en"> set the name of the Reader Device. </xsd:documentation>
6985   </xsd:annotation>
6986   <xsd:complexType>
6987     <xsd:sequence>
6988       <xsd:element name="name" type="xsd:string">
6989         <xsd:annotation>
6990           <xsd:documentation xml:lang="en"> new name to set. </xsd:documentation>
6991         </xsd:annotation>
6992       </xsd:element>
6993     </xsd:sequence>
6994   </xsd:complexType>
6995 </xsd:element>
6996 <xsd:element name="getRole" type="rp:noParamType">
6997   <xsd:annotation>
6998     <xsd:documentation xml:lang="en"> get the role of the Reader Device. </xsd:documentation>
6999   </xsd:annotation>
7000 </xsd:element>
7001 <xsd:element name="setRole">
7002   <xsd:annotation>
7003     <xsd:documentation xml:lang="en"> set the Handle of the Reader Device. </xsd:documentation>
7004   </xsd:annotation>
7005   <xsd:complexType>
7006     <xsd:sequence>
7007       <xsd:element name="role" type="xsd:string">
7008         <xsd:annotation>
7009           <xsd:documentation xml:lang="en"> Role to set. </xsd:documentation>
7010         </xsd:annotation>
7011       </xsd:element>
7012     </xsd:sequence>
7013   </xsd:complexType>
7014 </xsd:element>
7015 <xsd:element name="getTimeTicks" type="rp:noParamType">
7016   <xsd:annotation>
7017     <xsd:documentation xml:lang="en"> get the time in ticks at the Reader Device. </xsd:documentation>
7018   </xsd:annotation>
7019 </xsd:element>
7020 <xsd:element name="getTimeUTC" type="rp:noParamType">
7021   <xsd:annotation>
7022     <xsd:documentation xml:lang="en"> get the UTC time of the Reader Device. </xsd:documentation>
7023   </xsd:annotation>
7024 </xsd:element>
7025 <xsd:element name="setTimeUTC">
7026   <xsd:annotation>
7027     <xsd:documentation xml:lang="en"> set the UTC time of the Reader Device. </xsd:documentation>
7028   </xsd:annotation>
7029   <xsd:complexType>
7030     <xsd:sequence>
7031       <xsd:element name="utc" type="xsd:dateTime">
7032         <xsd:annotation>
7033           <xsd:documentation xml:lang="en"> UTC time to set. </xsd:documentation>
7034         </xsd:annotation>
7035       </xsd:element>
7036     </xsd:sequence>

```

```

7037     </xsd:complexType>
7038 </xsd:element>
7039 <xsd:element name="getManufacturerDescription" type="rp:noParamType">
7040     <xsd:annotation>
7041         <xsd:documentation>get the description of the reader as defined by the
7042 manufacturer</xsd:documentation>
7043     </xsd:annotation>
7044 </xsd:element>
7045 <xsd:element name="getCurrentSource" type="rp:noParamType">
7046     <xsd:annotation>
7047         <xsd:documentation xml:lang="en"> get the current source. </xsd:documentation>
7048     </xsd:annotation>
7049 </xsd:element>
7050 <xsd:element name="setCurrentSource">
7051     <xsd:annotation>
7052         <xsd:documentation xml:lang="en"> set the current source. </xsd:documentation>
7053     </xsd:annotation>
7054     <xsd:complexType>
7055         <xsd:sequence>
7056             <xsd:element name="currentSource" type="rp:SourceParamType">
7057                 <xsd:annotation>
7058                     <xsd:documentation xml:lang="en"> source to set the current source
7059 to. </xsd:documentation>
7060                 </xsd:annotation>
7061             </xsd:element>
7062         </xsd:sequence>
7063     </xsd:complexType>
7064 </xsd:element>
7065 <xsd:element name="getCurrentDataSelector" type="rp:noParamType">
7066     <xsd:annotation>
7067         <xsd:documentation xml:lang="en"> get the current data selector of the reader
7068 device. </xsd:documentation>
7069     </xsd:annotation>
7070 </xsd:element>
7071 <xsd:element name="setCurrentDataSelector">
7072     <xsd:annotation>
7073         <xsd:documentation xml:lang="en"> set the current data selector of the reader
7074 device. </xsd:documentation>
7075     </xsd:annotation>
7076     <xsd:complexType>
7077         <xsd:sequence>
7078             <xsd:element name="currentDataSelector" type="rp:DataSelectorParamType">
7079                 <xsd:annotation>
7080                     <xsd:documentation xml:lang="en"> data selector to set as current. </xsd:documentation>
7081                 </xsd:annotation>
7082             </xsd:element>
7083         </xsd:sequence>
7084     </xsd:complexType>
7085 </xsd:element>
7086 <xsd:element name="removeSources">
7087     <xsd:annotation>
7088         <xsd:documentation xml:lang="en"> remove sources from the reader device. </xsd:documentation>
7089     </xsd:annotation>
7090     <xsd:complexType>
7091         <xsd:sequence>

```

```

7092      <xsd:element name="sources" type="rp:SourceListParamType">
7093          <xsd:annotation>
7094              <xsd:documentation xml:lang="en"> sources to remove from the reader
7095                  device. </xsd:documentation>
7096          </xsd:annotation>
7097      </xsd:element>
7098  </xsd:sequence>
7099 </xsd:complexType>
7100 </xsd:element>
7101 <xsd:element name="removeAllSources" type="rp:noParamType">
7102     <xsd:annotation>
7103         <xsd:documentation xml:lang="en"> remove all the sources from the reader device. </xsd:documentation>
7104     </xsd:annotation>
7105 </xsd:element>
7106 <xsd:element name="getSource">
7107     <xsd:annotation>
7108         <xsd:documentation xml:lang="en"> get the named source from the reader device. </xsd:documentation>
7109     </xsd:annotation>
7110     <xsd:complexType>
7111         <xsd:sequence>
7112             <xsd:element name="name" type="xsd:string">
7113                 <xsd:annotation>
7114                     <xsd:documentation xml:lang="en"> name of the source to get. </xsd:documentation>
7115                 </xsd:annotation>
7116             </xsd:element>
7117         </xsd:sequence>
7118     </xsd:complexType>
7119 </xsd:element>
7120 <xsd:element name="getAllSources" type="rp:noParamType">
7121     <xsd:annotation>
7122         <xsd:documentation xml:lang="en"> get all the sources in the reader device. </xsd:documentation>
7123     </xsd:annotation>
7124 </xsd:element>
7125 <xsd:element name="removeDataSelectors">
7126     <xsd:annotation>
7127         <xsd:documentation xml:lang="en"> remove data selectors from the reader device. </xsd:documentation>
7128     </xsd:annotation>
7129     <xsd:complexType>
7130         <xsd:sequence>
7131             <xsd:element name="dataSelectors" type="rp:DataSelectorListParamType">
7132                 <xsd:annotation>
7133                     <xsd:documentation xml:lang="en"> data selector to remove. </xsd:documentation>
7134                 </xsd:annotation>
7135             </xsd:element>
7136         </xsd:sequence>
7137     </xsd:complexType>
7138 </xsd:element>
7139 <xsd:element name="removeAllDataSelectors" type="rp:noParamType">
7140     <xsd:annotation>
7141         <xsd:documentation xml:lang="en"> remove all the data selectors from the reader
7142             device. </xsd:documentation>
7143     </xsd:annotation>
7144 </xsd:element>
7145 <xsd:element name="getDataSelector">
7146     <xsd:annotation>

```

```

7147      <xsd:documentation xml:lang="en"> get the named data selector from the reader
7148      device. </xsd:documentation>
7149    </xsd:annotation>
7150    <xsd:complexType>
7151      <xsd:sequence>
7152        <xsd:element name="name" type="xsd:string">
7153          <xsd:annotation>
7154            <xsd:documentation xml:lang="en"> name of the data selector to get. </xsd:documentation>
7155          </xsd:annotation>
7156        </xsd:element>
7157      </xsd:sequence>
7158    </xsd:complexType>
7159  </xsd:element>
7160  <xsd:element name="getAllDataSelectors" type="rp:noParamType">
7161    <xsd:annotation>
7162      <xsd:documentation xml:lang="en"> get all the data selectors in the reader
7163      device. </xsd:documentation>
7164    </xsd:annotation>
7165  </xsd:element>
7166  <xsd:element name="removeNotificationChannels">
7167    <xsd:annotation>
7168      <xsd:documentation xml:lang="en"> remove notification channels from the reader
7169      device. </xsd:documentation>
7170    </xsd:annotation>
7171    <xsd:complexType>
7172      <xsd:sequence>
7173        <xsd:element name="channels" type="rp:NotificationChannelListParamType">
7174          <xsd:annotation>
7175            <xsd:documentation xml:lang="en"> channel to remove. </xsd:documentation>
7176          </xsd:annotation>
7177        </xsd:element>
7178      </xsd:sequence>
7179    </xsd:complexType>
7180  </xsd:element>
7181  <xsd:element name="removeAllNotificationChannels" type="rp:noParamType">
7182    <xsd:annotation>
7183      <xsd:documentation xml:lang="en"> remove all notification channels from the
7184      reader device. </xsd:documentation>
7185    </xsd:annotation>
7186  </xsd:element>
7187  <xsd:element name="getNotificationChannel">
7188    <xsd:annotation>
7189      <xsd:documentation xml:lang="en"> get named notification channel from the reader
7190      device. </xsd:documentation>
7191    </xsd:annotation>
7192    <xsd:complexType>
7193      <xsd:sequence>
7194        <xsd:element name="name" type="xsd:string">
7195          <xsd:annotation>
7196            <xsd:documentation xml:lang="en"> name of the channel to get. </xsd:documentation>
7197          </xsd:annotation>
7198        </xsd:element>
7199      </xsd:sequence>
7200    </xsd:complexType>
7201  </xsd:element>

```

```

7202 <xsd:element name="getAllNotificationChannels" type="rp:noParamType">
7203   <xsd:annotation>
7204     <xsd:documentation xml:lang="en"> get all the notification channels in the
7205       reader device. </xsd:documentation>
7206   </xsd:annotation>
7207 </xsd:element>
7208 <xsd:element name="removeTriggers">
7209   <xsd:annotation>
7210     <xsd:documentation>remove the triggers from the reader device.</xsd:documentation>
7211   </xsd:annotation>
7212   <xsd:complexType>
7213     <xsd:sequence>
7214       <xsd:element name="triggers" type="rp:TriggerListParamType">
7215         <xsd:annotation>
7216           <xsd:documentation>the triggers to be removed.</xsd:documentation>
7217         </xsd:annotation>
7218       </xsd:element>
7219     </xsd:sequence>
7220   </xsd:complexType>
7221 </xsd:element>
7222 <xsd:element name="removeAllTriggers" type="rp:noParamType">
7223   <xsd:annotation>
7224     <xsd:documentation>removes all triggers from the reader device.</xsd:documentation>
7225   </xsd:annotation>
7226 </xsd:element>
7227 <xsd:element name="getTrigger">
7228   <xsd:annotation>
7229     <xsd:documentation>get named trigger from the reader device</xsd:documentation>
7230   </xsd:annotation>
7231   <xsd:complexType>
7232     <xsd:sequence>
7233       <xsd:element name="name" type="xsd:string"/>
7234     </xsd:sequence>
7235   </xsd:complexType>
7236 </xsd:element>
7237 <xsd:element name="getAllTriggers" type="rp:noParamType">
7238   <xsd:annotation>
7239     <xsd:documentation>get all triggers</xsd:documentation>
7240   </xsd:annotation>
7241 </xsd:element>
7242 <xsd:element name="removeTagSelectors">
7243   <xsd:annotation>
7244     <xsd:documentation>remove the tag selectors from the reader device.</xsd:documentation>
7245   </xsd:annotation>
7246   <xsd:complexType>
7247     <xsd:sequence>
7248       <xsd:element name="selectors" type="rp:TagSelectorListParamType">
7249         <xsd:annotation>
7250           <xsd:documentation>the selectors to be removed.</xsd:documentation>
7251         </xsd:annotation>
7252       </xsd:element>
7253     </xsd:sequence>
7254   </xsd:complexType>
7255 </xsd:element>
7256 <xsd:element name="removeAllTagSelectors" type="rp:noParamType">

```

```

7257     <xsd:annotation>
7258         <xsd:documentation>removes all tag selectors from the reader device.</xsd:documentation>
7259     </xsd:annotation>
7260 </xsd:element>
7261 <xsd:element name="getTagSelector">
7262     <xsd:annotation>
7263         <xsd:documentation>get named TagSelector from the reader device</xsd:documentation>
7264     </xsd:annotation>
7265     <xsd:complexType>
7266         <xsd:sequence>
7267             <xsd:element name="name" type="xsd:string"/>
7268         </xsd:sequence>
7269     </xsd:complexType>
7270 </xsd:element>
7271 <xsd:element name="getAllTagSelectors" type="rp:noParamType">
7272     <xsd:annotation>
7273         <xsd:documentation>get all TagSelectors</xsd:documentation>
7274     </xsd:annotation>
7275 </xsd:element>
7276 <xsd:element name="removeTagFields">
7277     <xsd:annotation>
7278         <xsd:documentation>Removes the specified TagFields from the list of TagFields currently associated with
7279 the Reader</xsd:documentation>
7280     </xsd:annotation>
7281     <xsd:complexType>
7282         <xsd:sequence>
7283             <xsd:element name="fields" type="rp:TagFieldListParamType">
7284                 <xsd:annotation>
7285                     <xsd:documentation>the tag fields to be removed</xsd:documentation>
7286                 </xsd:annotation>
7287             </xsd:element>
7288         </xsd:sequence>
7289     </xsd:complexType>
7290 </xsd:element>
7291 <xsd:element name="removeAllTagFields" type="rp:noParamType">
7292     <xsd:annotation>
7293         <xsd:documentation>removes all tag fields</xsd:documentation>
7294     </xsd:annotation>
7295 </xsd:element>
7296 <xsd:element name="getTagField">
7297     <xsd:annotation>
7298         <xsd:documentation>get named tag field</xsd:documentation>
7299     </xsd:annotation>
7300     <xsd:complexType>
7301         <xsd:sequence>
7302             <xsd:element name="name" type="xsd:string"/>
7303         </xsd:sequence>
7304     </xsd:complexType>
7305 </xsd:element>
7306 <xsd:element name="getAllTagFields" type="rp:noParamType">
7307     <xsd:annotation>
7308         <xsd:documentation>get all tag fields</xsd:documentation>
7309     </xsd:annotation>
7310 </xsd:element>
7311 <xsd:element name="resetToDefaultSettings" type="rp:noParamType">

```

```

7312     <xsd:annotation>
7313         <xsd:documentation xml:lang="en"> reset the reader device to default settings. </xsd:documentation>
7314     </xsd:annotation>
7315 </xsd:element>
7316 <xsd:element name="reboot" type="rp:noParamType">
7317     <xsd:annotation>
7318         <xsd:documentation xml:lang="en"> reboot the reader device. </xsd:documentation>
7319     </xsd:annotation>
7320 </xsd:element>
7321 <xsd:element name="goodbye" type="rp:noParamType">
7322     <xsd:annotation>
7323         <xsd:documentation xml:lang="en"> closes this control channel. </xsd:documentation>
7324     </xsd:annotation>
7325 </xsd:element>
7326 <xsd:element name="getReadPoint">
7327     <xsd:annotation>
7328         <xsd:documentation>get the named read point</xsd:documentation>
7329     </xsd:annotation>
7330     <xsd:complexType>
7331         <xsd:sequence>
7332             <xsd:element name="name" type="xsd:string"/>
7333         </xsd:sequence>
7334     </xsd:complexType>
7335 </xsd:element>
7336 <xsd:element name="getAllReadPoints" type="rp:noParamType">
7337     <xsd:annotation>
7338         <xsd:documentation xml:lang="en"> get all the read points in the reader device. </xsd:documentation>
7339     </xsd:annotation>
7340 </xsd:element>
7341 <xsd:any namespace="##any" processContents="lax">
7342     <xsd:annotation>
7343         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
7344     </xsd:annotation>
7345 </xsd:any>
7346 </xsd:choice>
7347 </xsd:complexType>
7348 <xsd:complexType name="TagSelectorCommand">
7349     <xsd:choice>
7350         <xsd:annotation>
7351             <xsd:documentation xml:lang="en"> Read TagSelector object commands </xsd:documentation>
7352         </xsd:annotation>
7353         <xsd:element name="create">
7354             <xsd:annotation>
7355                 <xsd:documentation xml:lang="en"> static method to create a read TagSelector. </xsd:documentation>
7356             </xsd:annotation>
7357             <xsd:complexType>
7358                 <xsd:sequence>
7359                     <xsd:element name="name" type="xsd:string">
7360                         <xsd:annotation>
7361                             <xsd:documentation xml:lang="en"> name of the TagSelector. </xsd:documentation>
7362                         </xsd:annotation>
7363                     </xsd:element>
7364                     <xsd:element name="field" type="rp:TagFieldParamType">
7365                         <xsd:annotation>
7366                             <xsd:documentation>The name of the TagField</xsd:documentation>

```

```

7367         </xsd:annotation>
7368     </xsd:element>
7369     <xsd:element name="value" type="rp:HexStringType">
7370         <xsd:annotation>
7371             <xsd:documentation xml:lang="en"> value of the TagSelector. </xsd:documentation>
7372         </xsd:annotation>
7373     </xsd:element>
7374     <xsd:element name="mask" type="rp:HexStringType">
7375         <xsd:annotation>
7376             <xsd:documentation xml:lang="en"> mask of the TagSelector. </xsd:documentation>
7377         </xsd:annotation>
7378     </xsd:element>
7379     <xsd:element name="inclusiveFlag" type="xsd:boolean">
7380         <xsd:annotation>
7381             <xsd:documentation xml:lang="en"> whether the TagSelector is inclusive
7382             TagSelector. </xsd:documentation>
7383         </xsd:annotation>
7384     </xsd:element>
7385 </xsd:sequence>
7386 </xsd:complexType>
7387 </xsd:element>
7388 <xsd:element name="getMaxNumberSupported" type="rp:noParamType">
7389     <xsd:annotation>
7390         <xsd:documentation xml:lang="en"> get the maximum supported TagSelectors. </xsd:documentation>
7391     </xsd:annotation>
7392 </xsd:element>
7393 <xsd:element name="getTagField" type="rp:noParamType">
7394     <xsd:annotation>
7395         <xsd:documentation>get the tag field of the TagSelector</xsd:documentation>
7396     </xsd:annotation>
7397 </xsd:element>
7398 <xsd:element name="getName" type="rp:noParamType">
7399     <xsd:annotation>
7400         <xsd:documentation>get the name of the TagSelector</xsd:documentation>
7401     </xsd:annotation>
7402 </xsd:element>
7403 <xsd:element name="getValue" type="rp:noParamType">
7404     <xsd:annotation>
7405         <xsd:documentation xml:lang="en"> get the value of the TagSelector. </xsd:documentation>
7406     </xsd:annotation>
7407 </xsd:element>
7408 <xsd:element name="getMask" type="rp:noParamType">
7409     <xsd:annotation>
7410         <xsd:documentation xml:lang="en"> get the mask of the TagSelector. </xsd:documentation>
7411     </xsd:annotation>
7412 </xsd:element>
7413 <xsd:element name="getInclusiveFlag" type="rp:noParamType">
7414     <xsd:annotation>
7415         <xsd:documentation xml:lang="en"> return whether the TagSelector is inclusive TagSelector.
7416 </xsd:documentation>
7417     </xsd:annotation>
7418 </xsd:element>
7419 <xsd:any namespace="##any" processContents="lax">
7420     <xsd:annotation>
7421         <xsd:documentation>A standard or vendor extension</xsd:documentation>

```

```

7422     </xsd:annotation>
7423   </xsd:any>
7424 </xsd:choice>
7425 </xsd:complexType>
7426 <xsd:complexType name="ReadPointCommand">
7427   <xsd:choice>
7428     <xsd:element name="getName" type="rp:noParamType">
7429       <xsd:annotation>
7430         <xsd:documentation>get the name of the read point</xsd:documentation>
7431       </xsd:annotation>
7432     </xsd:element>
7433     <xsd:any namespace="##any" processContents="lax">
7434       <xsd:annotation>
7435         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
7436       </xsd:annotation>
7437     </xsd:any>
7438   </xsd:choice>
7439 </xsd:complexType>
7440 <xsd:complexType name="SourceCommand">
7441   <xsd:choice>
7442     <xsd:annotation>
7443       <xsd:documentation xml:lang="en"> Source object commands. </xsd:documentation>
7444     </xsd:annotation>
7445     <xsd:element name="create">
7446       <xsd:annotation>
7447         <xsd:documentation xml:lang="en"> static method to create a source. </xsd:documentation>
7448       </xsd:annotation>
7449     </xsd:element>
7450     <xsd:complexType>
7451       <xsd:sequence>
7452         <xsd:element name="name" type="xsd:string">
7453           <xsd:annotation>
7454             <xsd:documentation xml:lang="en"> name of the source. </xsd:documentation>
7455           </xsd:annotation>
7456         </xsd:element>
7457       </xsd:sequence>
7458     </xsd:complexType>
7459   </xsd:choice>
7460   <xsd:element name="getName" type="rp:noParamType">
7461     <xsd:annotation>
7462       <xsd:documentation>gets the name of the source</xsd:documentation>
7463     </xsd:annotation>
7464   </xsd:element>
7465   <xsd:element name="isFixed" type="rp:noParamType">
7466     <xsd:annotation>
7467       <xsd:documentation xml:lang="en"> get whether the source is fixed, e.g. ReadPoints can be added or not.
7468     </xsd:annotation>
7469   </xsd:element>
7470   <xsd:element name="addReadPoints">
7471     <xsd:annotation>
7472       <xsd:documentation xml:lang="en"> add read points to the source. </xsd:documentation>
7473     </xsd:annotation>
7474   </xsd:element>
7475   <xsd:complexType>
7476     <xsd:sequence>
7477       <xsd:element name="readPoints" type="rp:ReadPointListParamType">

```

```

7477         <xsd:annotation>
7478             <xsd:documentation xml:lang="en"> read points to add. </xsd:documentation>
7479         </xsd:annotation>
7480     </xsd:element>
7481 </xsd:sequence>
7482 </xsd:complexType>
7483 </xsd:element>
7484 <xsd:element name="removeReadPoints">
7485     <xsd:annotation>
7486         <xsd:documentation xml:lang="en"> remove read points from the source. </xsd:documentation>
7487     </xsd:annotation>
7488     <xsd:complexType>
7489         <xsd:sequence>
7490             <xsd:element name="readPoints" type="rp:ReadPointListParamType">
7491                 <xsd:annotation>
7492                     <xsd:documentation xml:lang="en"> read points to remove. </xsd:documentation>
7493                 </xsd:annotation>
7494             </xsd:element>
7495         </xsd:sequence>
7496     </xsd:complexType>
7497 </xsd:element>
7498 <xsd:element name="removeAllReadPoints" type="rp:noParamType">
7499     <xsd:annotation>
7500         <xsd:documentation xml:lang="en"> remove all the read points from the source. </xsd:documentation>
7501     </xsd:annotation>
7502 </xsd:element>
7503 <xsd:element name="getReadPoint">
7504     <xsd:annotation>
7505         <xsd:documentation xml:lang="en"> get the named read point from the source. </xsd:documentation>
7506     </xsd:annotation>
7507     <xsd:complexType>
7508         <xsd:sequence>
7509             <xsd:element name="name" type="xsd:string">
7510                 <xsd:annotation>
7511                     <xsd:documentation xml:lang="en"> name of the read point to get. </xsd:documentation>
7512                 </xsd:annotation>
7513             </xsd:element>
7514         </xsd:sequence>
7515     </xsd:complexType>
7516 </xsd:element>
7517 <xsd:element name="getAllReadPoints" type="rp:noParamType">
7518     <xsd:annotation>
7519         <xsd:documentation xml:lang="en"> get all read points from the source. </xsd:documentation>
7520     </xsd:annotation>
7521 </xsd:element>
7522 <xsd:element name="addReadTriggers">
7523     <xsd:annotation>
7524         <xsd:documentation xml:lang="en"> add read triggers to the source. </xsd:documentation>
7525     </xsd:annotation>
7526     <xsd:complexType>
7527         <xsd:sequence>
7528             <xsd:element name="triggers" type="rp:TriggerListParamType">
7529                 <xsd:annotation>
7530                     <xsd:documentation xml:lang="en"> triggers to add. </xsd:documentation>
7531                 </xsd:annotation>

```

```

7532         </xsd:element>
7533     </xsd:sequence>
7534 </xsd:complexType>
7535 </xsd:element>
7536 <xsd:element name="removeReadTriggers">
7537     <xsd:annotation>
7538         <xsd:documentation xml:lang="en"> remove read triggers from the source. </xsd:documentation>
7539     </xsd:annotation>
7540     <xsd:complexType>
7541         <xsd:sequence>
7542             <xsd:element name="triggers" type="rp:TriggerListParamType">
7543                 <xsd:annotation>
7544                     <xsd:documentation xml:lang="en"> read triggers to remove. </xsd:documentation>
7545                 </xsd:annotation>
7546             </xsd:element>
7547         </xsd:sequence>
7548     </xsd:complexType>
7549 </xsd:element>
7550 <xsd:element name="removeAllReadTriggers" type="rp:noParamType">
7551     <xsd:annotation>
7552         <xsd:documentation xml:lang="en"> remove all read triggers from the source. </xsd:documentation>
7553     </xsd:annotation>
7554 </xsd:element>
7555 <xsd:element name="getReadTrigger">
7556     <xsd:annotation>
7557         <xsd:documentation xml:lang="en"> get the named read trigger. </xsd:documentation>
7558     </xsd:annotation>
7559     <xsd:complexType>
7560         <xsd:sequence>
7561             <xsd:element name="name" type="xsd:string">
7562                 <xsd:annotation>
7563                     <xsd:documentation xml:lang="en"> name of the read trigger to get. </xsd:documentation>
7564                 </xsd:annotation>
7565             </xsd:element>
7566         </xsd:sequence>
7567     </xsd:complexType>
7568 </xsd:element>
7569 <xsd:element name="getAllReadTriggers" type="rp:noParamType">
7570     <xsd:annotation>
7571         <xsd:documentation xml:lang="en"> get all the read triggers in the source. </xsd:documentation>
7572     </xsd:annotation>
7573 </xsd:element>
7574 <xsd:element name="addTagSelectors">
7575     <xsd:annotation>
7576         <xsd:documentation xml:lang="en"> add read TagSelectors to the source. </xsd:documentation>
7577     </xsd:annotation>
7578     <xsd:complexType>
7579         <xsd:sequence>
7580             <xsd:element name="selectors" type="rp:TagSelectorListParamType">
7581                 <xsd:annotation>
7582                     <xsd:documentation xml:lang="en"> TagSelectors to add. </xsd:documentation>
7583                 </xsd:annotation>
7584             </xsd:element>
7585         </xsd:sequence>
7586     </xsd:complexType>

```

```

7587 </xsd:element>
7588 <xsd:element name="removeTagSelectors">
7589   <xsd:annotation>
7590     <xsd:documentation xml:lang="en"> remove read TagSelectors from the source. </xsd:documentation>
7591   </xsd:annotation>
7592   <xsd:complexType>
7593     <xsd:sequence>
7594       <xsd:element name="selectors" type="rp:TagSelectorListParamType">
7595         <xsd:annotation>
7596           <xsd:documentation xml:lang="en"> tag selectors to remove. </xsd:documentation>
7597         </xsd:annotation>
7598       </xsd:element>
7599     </xsd:sequence>
7600   </xsd:complexType>
7601 </xsd:element>
7602 <xsd:element name="removeAllTagSelectors" type="rp:noParamType">
7603   <xsd:annotation>
7604     <xsd:documentation xml:lang="en"> remove all tag selectors from the source. </xsd:documentation>
7605   </xsd:annotation>
7606 </xsd:element>
7607 <xsd:element name="getTagSelector">
7608   <xsd:annotation>
7609     <xsd:documentation xml:lang="en"> get the named tag selector from the source. </xsd:documentation>
7610   </xsd:annotation>
7611   <xsd:complexType>
7612     <xsd:sequence>
7613       <xsd:element name="name" type="xsd:string">
7614         <xsd:annotation>
7615           <xsd:documentation xml:lang="en"> name of the tag selector to get. </xsd:documentation>
7616         </xsd:annotation>
7617       </xsd:element>
7618     </xsd:sequence>
7619   </xsd:complexType>
7620 </xsd:element>
7621 <xsd:element name="getAllTagSelectors" type="rp:noParamType">
7622   <xsd:annotation>
7623     <xsd:documentation xml:lang="en"> get all the tag selector in the source. </xsd:documentation>
7624   </xsd:annotation>
7625 </xsd:element>
7626 <xsd:element name="getGlimpsedTimeout" type="rp:noParamType">
7627   <xsd:annotation>
7628     <xsd:documentation xml:lang="en"> get Timeout in ms before it moves from
7629       isGlimpsed state to isUnknown state. </xsd:documentation>
7630   </xsd:annotation>
7631 </xsd:element>
7632 <xsd:element name="setGlimpsedTimeout">
7633   <xsd:annotation>
7634     <xsd:documentation xml:lang="en"> set Timeout in ms before it moves from
7635       isGlimpsed state to isUnknown state. </xsd:documentation>
7636   </xsd:annotation>
7637   <xsd:complexType>
7638     <xsd:sequence>
7639       <xsd:element name="timeout" type="xsd:int">
7640         <xsd:annotation>
7641           <xsd:documentation xml:lang="en"> time in ms before it moves from

```

```

7642         isGlimpsed state to isUnknown state. </xsd:documentation>
7643     </xsd:annotation>
7644 </xsd:element>
7645 </xsd:sequence>
7646 </xsd:complexType>
7647 </xsd:element>
7648 <xsd:element name="getObservedThreshold" type="rp:noParamType">
7649     <xsd:annotation>
7650         <xsd:documentation xml:lang="en"> get time in ms after which it moves from
7651             isGlimpsed state to isObserved state. </xsd:documentation>
7652     </xsd:annotation>
7653 </xsd:element>
7654 <xsd:element name="setObservedThreshold">
7655     <xsd:annotation>
7656         <xsd:documentation xml:lang="en"> set time in ms after which it moves from
7657
7658             isGlimpsed state to isObserved state. </xsd:documentation>
7659     </xsd:annotation>
7660 <xsd:complexType>
7661     <xsd:sequence>
7662         <xsd:element name="threshold" type="xsd:int">
7663             <xsd:annotation>
7664                 <xsd:documentation xml:lang="en"> time in ms after which it moves
7665                     from isGlimpsed state to isObserved state. </xsd:documentation>
7666             </xsd:annotation>
7667         </xsd:element>
7668     </xsd:sequence>
7669 </xsd:complexType>
7670 </xsd:element>
7671 <xsd:element name="getObservedTimeout" type="rp:noParamType">
7672     <xsd:annotation>
7673         <xsd:documentation xml:lang="en"> get time in ms after which it moves from
7674             isObserved state to isLost state. </xsd:documentation>
7675     </xsd:annotation>
7676 </xsd:element>
7677 <xsd:element name="setObservedTimeout">
7678     <xsd:annotation>
7679         <xsd:documentation xml:lang="en"> set time in ms after which it moves from
7680             isObserved state to isLost state. </xsd:documentation>
7681     </xsd:annotation>
7682 <xsd:complexType>
7683     <xsd:sequence>
7684         <xsd:element name="timeout" type="xsd:int">
7685             <xsd:annotation>
7686                 <xsd:documentation xml:lang="en"> time in ms after which it moves
7687                     from isObserved state to isLost state. </xsd:documentation>
7688             </xsd:annotation>
7689         </xsd:element>
7690     </xsd:sequence>
7691 </xsd:complexType>
7692 </xsd:element>
7693 <xsd:element name="getLostTimeout" type="rp:noParamType">
7694     <xsd:annotation>
7695         <xsd:documentation xml:lang="en"> get time in ms after which it moves from
7696             isLost state to isUnknown state. </xsd:documentation>

```

```

7697     </xsd:annotation>
7698 </xsd:element>
7699 <xsd:element name="setLostTimeout">
7700     <xsd:annotation>
7701         <xsd:documentation xml:lang="en"> set time in ms after which it moves from
7702             isLost state to isUnknown state. </xsd:documentation>
7703     </xsd:annotation>
7704     <xsd:complexType>
7705         <xsd:sequence>
7706             <xsd:element name="timeout" type="xsd:int">
7707                 <xsd:annotation>
7708                     <xsd:documentation xml:lang="en"> time in ms after which it moves
7709                         from isLost state to isUnknown state. </xsd:documentation>
7710                 </xsd:annotation>
7711             </xsd:element>
7712         </xsd:sequence>
7713     </xsd:complexType>
7714 </xsd:element>
7715 <xsd:element name="rawReadIDs">
7716     <xsd:annotation>
7717         <xsd:documentation xml:lang="en"> returns ReadReport with raw(unprocessed) read
7718             lds. </xsd:documentation>
7719     </xsd:annotation>
7720     <xsd:complexType>
7721         <xsd:sequence>
7722             <xsd:element name="dataSelector" type="rp:DataSelectorParamType" minOccurs="0">
7723                 <xsd:annotation>
7724                     <xsd:documentation xml:lang="en"> Data selector to apply to lds
7725                         read. </xsd:documentation>
7726                 </xsd:annotation>
7727             </xsd:element>
7728         </xsd:sequence>
7729     </xsd:complexType>
7730 </xsd:element>
7731 <xsd:element name="readIDs">
7732     <xsd:annotation>
7733         <xsd:documentation xml:lang="en"> returns ReadReport with read lds. </xsd:documentation>
7734     </xsd:annotation>
7735     <xsd:complexType>
7736         <xsd:sequence>
7737             <xsd:element name="dataSelector" type="rp:DataSelectorParamType" minOccurs="0">
7738                 <xsd:annotation>
7739                     <xsd:documentation xml:lang="en"> Data selector to apply to lds
7740                         read. </xsd:documentation>
7741                 </xsd:annotation>
7742             </xsd:element>
7743         </xsd:sequence>
7744     </xsd:complexType>
7745 </xsd:element>
7746 <xsd:element name="read">
7747     <xsd:annotation>
7748         <xsd:documentation>reads tag memory</xsd:documentation>
7749     </xsd:annotation>
7750     <xsd:complexType>
7751         <xsd:sequence>

```

```

7752     <xsd:element name="dataSelector" type="rp:DataSelectorParamType"/>
7753     <xsd:element name="passwords" type="rp:HexStringListType" minOccurs="0"/>
7754     <xsd:element name="selectors" type="rp:TagSelectorListParamType" minOccurs="0"/>
7755   </xsd:sequence>
7756 </xsd:complexType>
7757 </xsd:element>
7758 <xsd:element name="writeID">
7759   <xsd:annotation>
7760     <xsd:documentation xml:lang="en"> write id to the tag </xsd:documentation>
7761   </xsd:annotation>
7762   <xsd:complexType>
7763     <xsd:sequence>
7764       <xsd:element name="id" type="xsd:hexBinary">
7765         <xsd:annotation>
7766           <xsd:documentation xml:lang="en"> id to write. </xsd:documentation>
7767         </xsd:annotation>
7768       </xsd:element>
7769       <xsd:element name="passwords" type="rp:HexStringListType" minOccurs="0">
7770         <xsd:annotation>
7771           <xsd:documentation xml:lang="en"> passcode to write (optional). </xsd:documentation>
7772         </xsd:annotation>
7773       </xsd:element>
7774       <xsd:element name="selectors" type="rp:TagSelectorListParamType" minOccurs="0"/>
7775     </xsd:sequence>
7776   </xsd:complexType>
7777 </xsd:element>
7778 <xsd:element name="write">
7779   <xsd:complexType>
7780     <xsd:sequence>
7781       <xsd:element name="data" type="rp:TagFieldValueListParamType"/>
7782       <xsd:element name="passwords" type="rp:HexStringListType" minOccurs="0"/>
7783       <xsd:element name="selectors" type="rp:TagSelectorListParamType" minOccurs="0"/>
7784     </xsd:sequence>
7785   </xsd:complexType>
7786 </xsd:element>
7787 <xsd:element name="kill">
7788   <xsd:annotation>
7789     <xsd:documentation xml:lang="en"> kill a tag. </xsd:documentation>
7790   </xsd:annotation>
7791   <xsd:complexType>
7792     <xsd:sequence>
7793       <xsd:element name="passwords" type="rp:HexStringListType" minOccurs="0">
7794         <xsd:annotation>
7795           <xsd:documentation xml:lang="en"> passcode if required. </xsd:documentation>
7796         </xsd:annotation>
7797       </xsd:element>
7798       <xsd:element name="selectors" type="rp:TagSelectorListParamType" minOccurs="0">
7799         <xsd:annotation>
7800           <xsd:documentation xml:lang="en"> id of the tag to kill. </xsd:documentation>
7801         </xsd:annotation>
7802       </xsd:element>
7803     </xsd:sequence>
7804   </xsd:complexType>
7805 </xsd:element>
7806 <xsd:element name="getReadCyclesPerTrigger" type="rp:noParamType">

```

```

7807     <xsd:annotation>
7808         <xsd:documentation xml:lang="en"> gets the read cycles per trigger </xsd:documentation>
7809     </xsd:annotation>
7810 </xsd:element>
7811 <xsd:element name="setReadCyclesPerTrigger">
7812     <xsd:annotation>
7813         <xsd:documentation xml:lang="en">sets the read cycles per trigger </xsd:documentation>
7814     </xsd:annotation>
7815     <xsd:complexType>
7816         <xsd:sequence>
7817             <xsd:element name="cycles" type="xsd:int">
7818                 <xsd:annotation>
7819                     <xsd:documentation xml:lang="en"> read cycles per trigger </xsd:documentation>
7820                 </xsd:annotation>
7821             </xsd:element>
7822         </xsd:sequence>
7823     </xsd:complexType>
7824 </xsd:element>
7825 <xsd:element name="getMaxReadDutyCycle" type="rp:noParamType">
7826     <xsd:annotation>
7827         <xsd:documentation xml:lang="en"> get read duty cycle percentage</xsd:documentation>
7828     </xsd:annotation>
7829 </xsd:element>
7830 <xsd:element name="setMaxReadDutyCycle">
7831     <xsd:annotation>
7832         <xsd:documentation xml:lang="en">sets the read duty cycle percentage</xsd:documentation>
7833     </xsd:annotation>
7834     <xsd:complexType>
7835         <xsd:sequence>
7836             <xsd:element name="dutyCycle" type="xsd:int">
7837                 <xsd:annotation>
7838                     <xsd:documentation xml:lang="en"> read duty cycle percentage</xsd:documentation>
7839                 </xsd:annotation>
7840             </xsd:element>
7841         </xsd:sequence>
7842     </xsd:complexType>
7843 </xsd:element>
7844 <xsd:element name="getReadTimeout" type="rp:noParamType">
7845     <xsd:annotation>
7846         <xsd:documentation xml:lang="en"> get read timeout</xsd:documentation>
7847     </xsd:annotation>
7848 </xsd:element>
7849 <xsd:element name="setReadTimeout">
7850     <xsd:annotation>
7851         <xsd:documentation xml:lang="en">set read timeout</xsd:documentation>
7852     </xsd:annotation>
7853     <xsd:complexType>
7854         <xsd:sequence>
7855             <xsd:element name="timeout" type="xsd:int">
7856                 <xsd:annotation>
7857                     <xsd:documentation xml:lang="en">read timeout</xsd:documentation>
7858                 </xsd:annotation>
7859             </xsd:element>
7860         </xsd:sequence>
7861     </xsd:complexType>

```

```

7862 </xsd:element>
7863 <xsd:element name="getSession" type="rp:noParamType">
7864   <xsd:annotation>
7865     <xsd:documentation>get the (RF-protocol-dependent) session attribute</xsd:documentation>
7866   </xsd:annotation>
7867 </xsd:element>
7868 <xsd:element name="setSession">
7869   <xsd:annotation>
7870     <xsd:documentation>set the (RF-protocol-dependent) session attribute</xsd:documentation>
7871   </xsd:annotation>
7872   <xsd:complexType>
7873     <xsd:sequence>
7874       <xsd:element name="session" type="xsd:int"/>
7875     </xsd:sequence>
7876   </xsd:complexType>
7877 </xsd:element>
7878 <xsd:any namespace="###any" processContents="lax">
7879   <xsd:annotation>
7880     <xsd:documentation>For standard and vendor extensions</xsd:documentation>
7881   </xsd:annotation>
7882 </xsd:any>
7883 </xsd:choice>
7884 </xsd:complexType>
7885 <xsd:complexType name="TriggerCommand">
7886   <xsd:choice>
7887     <xsd:annotation>
7888       <xsd:documentation xml:lang="en"> Trigger object commands. </xsd:documentation>
7889     </xsd:annotation>
7890     <xsd:element name="create">
7891       <xsd:annotation>
7892         <xsd:documentation xml:lang="en"> creates a Trigger. </xsd:documentation>
7893       </xsd:annotation>
7894       <xsd:complexType>
7895         <xsd:sequence>
7896           <xsd:element name="name" type="xsd:string">
7897             <xsd:annotation>
7898               <xsd:documentation xml:lang="en"> Trigger name. </xsd:documentation>
7899             </xsd:annotation>
7900           </xsd:element>
7901           <xsd:element name="triggerType" type="rp:TriggerTypeParamType">
7902             <xsd:annotation>
7903               <xsd:documentation xml:lang="en"> Trigger type. </xsd:documentation>
7904             </xsd:annotation>
7905           </xsd:element>
7906           <xsd:element name="triggerValue" type="rp:TriggerValueParamType">
7907             <xsd:annotation>
7908               <xsd:documentation xml:lang="en"> Trigger value. </xsd:documentation>
7909             </xsd:annotation>
7910           </xsd:element>
7911         </xsd:sequence>
7912       </xsd:complexType>
7913     </xsd:element>
7914     <xsd:element name="getMaxNumberSupported" type="rp:noParamType">
7915       <xsd:annotation>
7916         <xsd:documentation>get the maximum number of triggers supported</xsd:documentation>

```

```

7917     </xsd:annotation>
7918 </xsd:element>
7919 <xsd:element name="getName" type="rp:noParamType">
7920     <xsd:annotation>
7921         <xsd:documentation>gets the name of the trigger</xsd:documentation>
7922     </xsd:annotation>
7923 </xsd:element>
7924 <xsd:element name="getType" type="rp:noParamType">
7925     <xsd:annotation>
7926         <xsd:documentation xml:lang="en"> get the Trigger type. </xsd:documentation>
7927     </xsd:annotation>
7928 </xsd:element>
7929 <xsd:element name="getValue" type="rp:noParamType">
7930     <xsd:annotation>
7931         <xsd:documentation xml:lang="en"> get the trigger value. </xsd:documentation>
7932     </xsd:annotation>
7933 </xsd:element>
7934 <xsd:element name="fire" type="rp:noParamType">
7935     <xsd:annotation>
7936         <xsd:documentation xml:lang="en"> fire the trigger. </xsd:documentation>
7937     </xsd:annotation>
7938 </xsd:element>
7939 <xsd:any namespace="##any" processContents="lax">
7940     <xsd:annotation>
7941         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
7942     </xsd:annotation>
7943 </xsd:any>
7944 </xsd:choice>
7945 </xsd:complexType>
7946 <xsd:complexType name="TriggerTypeCommand">
7947     <xsd:annotation>
7948         <xsd:documentation xml:lang="en"> Trigger type enumeration object commands. </xsd:documentation>
7949     </xsd:annotation>
7950 <xsd:choice>
7951     <xsd:element name="getSupportedTypes" type="rp:noParamType">
7952         <xsd:annotation>
7953             <xsd:documentation xml:lang="en"> get all the trigger types supported in the
7954                 reader device. </xsd:documentation>
7955         </xsd:annotation>
7956     </xsd:element>
7957 </xsd:choice>
7958 </xsd:complexType>
7959 <xsd:complexType name="TagFieldCommand">
7960     <xsd:choice>
7961     <xsd:element name="create">
7962         <xsd:annotation>
7963             <xsd:documentation>creates a tag field</xsd:documentation>
7964         </xsd:annotation>
7965     <xsd:complexType>
7966         <xsd:sequence>
7967             <xsd:element name="name" type="xsd:string">
7968                 <xsd:annotation>
7969                     <xsd:documentation>the name of the object</xsd:documentation>
7970                 </xsd:annotation>
7971             </xsd:element>

```

```

7972         </xsd:sequence>
7973     </xsd:complexType>
7974 </xsd:element>
7975 <xsd:element name="getName" type="rp:noParamType">
7976     <xsd:annotation>
7977         <xsd:documentation>returns the name of this TagField object</xsd:documentation>
7978     </xsd:annotation>
7979 </xsd:element>
7980 <xsd:element name="getTagFieldName" type="rp:noParamType">
7981     <xsd:annotation>
7982         <xsd:documentation>returns the name of the data field this object is used for</xsd:documentation>
7983     </xsd:annotation>
7984 </xsd:element>
7985 <xsd:element name="setTagFieldName">
7986     <xsd:annotation>
7987         <xsd:documentation>sets the name of the data field this object is used for</xsd:documentation>
7988     </xsd:annotation>
7989 <xsd:complexType>
7990     <xsd:sequence>
7991         <xsd:element name="tagFieldName" type="xsd:string"/>
7992     </xsd:sequence>
7993 </xsd:complexType>
7994 </xsd:element>
7995 <xsd:element name="getMemoryBank" type="rp:noParamType">
7996     <xsd:annotation>
7997         <xsd:documentation>returns the memory bank where the data is stored on the tag</xsd:documentation>
7998     </xsd:annotation>
7999 </xsd:element>
8000 <xsd:element name="setMemoryBank">
8001     <xsd:annotation>
8002         <xsd:documentation>sets the memory bank where the data is stored on the tag</xsd:documentation>
8003     </xsd:annotation>
8004 <xsd:complexType>
8005     <xsd:sequence>
8006         <xsd:element name="memoryBank" type="xsd:integer"/>
8007     </xsd:sequence>
8008 </xsd:complexType>
8009 </xsd:element>
8010 <xsd:element name="getOffset" type="rp:noParamType">
8011     <xsd:annotation>
8012         <xsd:documentation>returns the address offset where the data is stored on the tag</xsd:documentation>
8013     </xsd:annotation>
8014 </xsd:element>
8015 <xsd:element name="setOffset">
8016     <xsd:annotation>
8017         <xsd:documentation>sets the address offset where the data is stored on the tag</xsd:documentation>
8018     </xsd:annotation>
8019 <xsd:complexType>
8020     <xsd:sequence>
8021         <xsd:element name="offset" type="xsd:int"/>
8022     </xsd:sequence>
8023 </xsd:complexType>
8024 </xsd:element>
8025 <xsd:element name="getLength" type="rp:noParamType">
8026     <xsd:annotation>

```

```

8027         <xsd:documentation>returns the length of this data field</xsd:documentation>
8028     </xsd:annotation>
8029 </xsd:element>
8030 <xsd:element name="setLength">
8031     <xsd:annotation>
8032         <xsd:documentation>sets the length of this data field</xsd:documentation>
8033     </xsd:annotation>
8034     <xsd:complexType>
8035         <xsd:sequence>
8036             <xsd:element name="length" type="xsd:int"/>
8037         </xsd:sequence>
8038     </xsd:complexType>
8039 </xsd:element>
8040 </xsd:choice>
8041 </xsd:complexType>
8042 </xsd:schema>

```

A.3 Reply Schema

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="urn:epcglobal:rp:xsd:1" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:epcglobal="urn:epcglobal:xsd:1" xmlns:rp="urn:epcglobal:rp:xsd:1" elementFormDefault="unqualified"
attributeFormDefault="unqualified" version="1.0">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      <epcglobal:copyright>Copyright ©2003-2006 Epcglobal Inc., All Rights Reserved.</epcglobal:copyright>
      <epcglobal:disclaimer>EPCglobal Inc., its members, officers, directors, employees, or
        agents shall not be liable for any injury, loss, damages, financial or otherwise,
        arising from, related to, or caused by the use of this document. The use of said
        document shall constitute your express consent to the foregoing exculpation.</epcglobal:disclaimer>
      <epcglobal:specification>Reader Protocol (RP) version 1.1</epcglobal:specification>
    </xsd:documentation>
  </xsd:annotation>
  <xsd:include schemaLocation="RpCommon.xsd"/>
  <xsd:import namespace="urn:epcglobal:xsd:1" schemaLocation="./EpcGlobal.xsd"/>
  <!-- Reader Protocol Reply -->
  <xsd:element name="reply">
    <xsd:annotation>
      <xsd:documentation xml:lang="en"> This element defines a reader protocol reply. Just
        like commands, replies are grouped by the object they belong to. To specify the
        object a reply relates to, the name of the object is used. All the replies contain a
        single return value inside the reply. return value is enclosed by the element
        "returnValue". In case of a single return value it is directly added to this
        element. In case of an array of return values they are enclosed within element
        "list" with each value in turn enclosed in element "value". Each reply is wrapped in
        its object type. </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="id" type="xsd:string">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">The id of the command for which
              this is the reply.</xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="resultCode" type="xsd:int">
          <xsd:annotation>
            <xsd:documentation xml:lang="en"> the error code for the command - 0
              for success.</xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:choice>
          <xsd:element name="error" type="rp:ErrorType">
            <xsd:annotation>
              <xsd:documentation xml:lang="en">Information about the error.</xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="readerDevice" type="rp:ReaderDeviceReply">
            <xsd:annotation>
              <xsd:documentation xml:lang="en">A Reader</xsd:documentation>
            </xsd:annotation>
          </xsd:element>
        </xsd:choice>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

```

```

8097 </xsd:element>
8098 <xsd:element name="source" type="rp:SourceReply">
8099   <xsd:annotation>
8100     <xsd:documentation xml:lang="en">A Read Source</xsd:documentation>
8101   </xsd:annotation>
8102 </xsd:element>
8103 <xsd:element name="readPoint" type="rp:ReadPointReply">
8104   <xsd:annotation>
8105     <xsd:documentation>A read point</xsd:documentation>
8106   </xsd:annotation>
8107 </xsd:element>
8108 <xsd:element name="trigger" type="rp:TriggerReply">
8109   <xsd:annotation>
8110     <xsd:documentation xml:lang="en">A Read/Notify Trigger</xsd:documentation>
8111   </xsd:annotation>
8112 </xsd:element>
8113 <xsd:element name="tagSelector" type="rp:TagSelectorReply">
8114   <xsd:annotation>
8115     <xsd:documentation xml:lang="en">A tag selector</xsd:documentation>
8116   </xsd:annotation>
8117 </xsd:element>
8118 <xsd:element name="notificationChannel" type="rp:NotificationChannelReply">
8119   <xsd:annotation>
8120     <xsd:documentation xml:lang="en">The notification channel
8121       carries messages issued asynchronously by the Reader to the Host.</xsd:documentation>
8122   </xsd:annotation>
8123 </xsd:element>
8124 <xsd:element name="dataSelector" type="rp:DataSelectorReply">
8125   <xsd:annotation>
8126     <xsd:documentation xml:lang="en">Which data is reported by (a
8127       notification channel/read command) depends on the settings
8128       of the associated DataSelector.</xsd:documentation>
8129   </xsd:annotation>
8130 </xsd:element>
8131 <xsd:element name="eventType" type="rp:EventTypeReply">
8132   <xsd:annotation>
8133     <xsd:documentation xml:lang="en">An enumeration object - Type of
8134       the Tag Event e.g. evGlimpsed</xsd:documentation>
8135   </xsd:annotation>
8136 </xsd:element>
8137 <xsd:element name="triggerType" type="rp:TriggerTypeReply">
8138   <xsd:annotation>
8139     <xsd:documentation xml:lang="en">An enumeration object - Type of
8140       the Read/Notify Trigger e.g. Continuous</xsd:documentation>
8141   </xsd:annotation>
8142 </xsd:element>
8143 <xsd:element name="fieldName" type="rp:FieldNameReply">
8144   <xsd:annotation>
8145     <xsd:documentation xml:lang="en">Name of the fields in a read report</xsd:documentation>
8146   </xsd:annotation>
8147 </xsd:element>
8148 <xsd:element name="tagField" type="rp:TagFieldReply">
8149   <xsd:annotation>
8150     <xsd:documentation>The data fields on a tag that can be read or written</xsd:documentation>
8151   </xsd:annotation>

```

```

8152     </xsd:element>
8153     <xsd:any namespace="##any" processContents="lax">
8154         <xsd:annotation>
8155             <xsd:documentation>A standard or vendor extension</xsd:documentation>
8156         </xsd:annotation>
8157     </xsd:any>
8158     <!-- Error information -->
8159     <!-- Reader Protocol (Target) Object Types -->
8160     <!-- elements inside are ordered in alphabetical order -->
8161     </xsd:choice>
8162 </xsd:sequence>
8163 </xsd:complexType>
8164 </xsd:element>
8165 <xsd:complexType name="ErrorType">
8166     <xsd:annotation>
8167         <xsd:documentation xml:lang="en">Information about the error.</xsd:documentation>
8168     </xsd:annotation>
8169     <xsd:sequence>
8170         <xsd:element name="name" type="xsd:string" minOccurs="0">
8171             <xsd:annotation>
8172                 <xsd:documentation xml:lang="en">Name of the error.</xsd:documentation>
8173             </xsd:annotation>
8174         </xsd:element>
8175         <xsd:element name="cause" type="xsd:string" minOccurs="0">
8176             <xsd:annotation>
8177                 <xsd:documentation xml:lang="en">Cause of the error.</xsd:documentation>
8178             </xsd:annotation>
8179         </xsd:element>
8180         <xsd:element name="description" type="xsd:string" minOccurs="0">
8181             <xsd:annotation>
8182                 <xsd:documentation xml:lang="en">Description of the error.</xsd:documentation>
8183             </xsd:annotation>
8184         </xsd:element>
8185         <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
8186     </xsd:sequence>
8187 </xsd:complexType>
8188 <!-- Reader Protocol (Target) Object Types -->
8189 <!-- types are ordered in alphabetical order -->
8190 <!-- inside each type replies are listed in the order in which it is defined in the protocol specification. -->
8191 <xsd:complexType name="DataSelectorReply">
8192     <xsd:choice>
8193         <xsd:annotation>
8194             <xsd:documentation xml:lang="en">Data selector object replies</xsd:documentation>
8195         </xsd:annotation>
8196         <xsd:element name="create" type="rp:DataSelectorReturnType">
8197             <xsd:annotation>
8198                 <xsd:documentation xml:lang="en">reply for create a data selector. </xsd:documentation>
8199             </xsd:annotation>
8200         </xsd:element>
8201         <xsd:element name="getName" type="rp:stringReturnType">
8202             <xsd:annotation>
8203                 <xsd:documentation>reply for get name</xsd:documentation>
8204             </xsd:annotation>
8205         </xsd:element>
8206         <xsd:element name="addFieldNames" type="rp:noParamType">

```

```

8207     <xsd:annotation>
8208         <xsd:documentation xml:lang="en"> reply for add FieldNames. </xsd:documentation>
8209     </xsd:annotation>
8210 </xsd:element>
8211 <xsd:element name="removeFieldNames" type="rp:noParamType">
8212     <xsd:annotation>
8213         <xsd:documentation xml:lang="en"> reply for remove FieldNames. </xsd:documentation>
8214     </xsd:annotation>
8215 </xsd:element>
8216 <xsd:element name="removeAllFieldNames" type="rp:noParamType">
8217     <xsd:annotation>
8218         <xsd:documentation xml:lang="en"> reply for remove All FieldNames. </xsd:documentation>
8219     </xsd:annotation>
8220 </xsd:element>
8221 <xsd:element name="getAllFieldNames" type="rp:FieldNameListReturnType">
8222     <xsd:annotation>
8223         <xsd:documentation xml:lang="en"> reply for get All FieldNames. </xsd:documentation>
8224     </xsd:annotation>
8225 </xsd:element>
8226 <xsd:element name="addEventFilters" type="rp:noParamType">
8227     <xsd:annotation>
8228         <xsd:documentation xml:lang="en"> reply for add EventFilters. </xsd:documentation>
8229     </xsd:annotation>
8230 </xsd:element>
8231 <xsd:element name="removeEventFilters" type="rp:noParamType">
8232     <xsd:annotation>
8233         <xsd:documentation xml:lang="en"> reply for remove EventFilters. </xsd:documentation>
8234     </xsd:annotation>
8235 </xsd:element>
8236 <xsd:element name="removeAllEventFilters" type="rp:noParamType">
8237     <xsd:annotation>
8238         <xsd:documentation xml:lang="en"> reply for remove all EventFilters. </xsd:documentation>
8239     </xsd:annotation>
8240 </xsd:element>
8241 <xsd:element name="getAllEventFilters" type="rp:EventTypeListReturnValue">
8242     <xsd:annotation>
8243         <xsd:documentation xml:lang="en"> reply for get All EventFilters. </xsd:documentation>
8244     </xsd:annotation>
8245 </xsd:element>
8246 <xsd:element name="addTagFieldNames" type="rp:noParamType">
8247     <xsd:annotation>
8248         <xsd:documentation>reply to addTagFieldNames</xsd:documentation>
8249     </xsd:annotation>
8250 </xsd:element>
8251 <xsd:element name="removeTagFieldNames" type="rp:noParamType">
8252     <xsd:annotation>
8253         <xsd:documentation>reply to removeTagFieldNames</xsd:documentation>
8254     </xsd:annotation>
8255 </xsd:element>
8256 <xsd:element name="removeAllTagFieldNames" type="rp:noParamType">
8257     <xsd:annotation>
8258         <xsd:documentation>reply to removeAllTagFieldNames</xsd:documentation>
8259     </xsd:annotation>
8260 </xsd:element>
8261 <xsd:element name="getAllTagFieldNames" type="rp:StringListReturnType">

```

```

8262     <xsd:annotation>
8263         <xsd:documentation>reply to getAllTagFieldNames</xsd:documentation>
8264     </xsd:annotation>
8265 </xsd:element>
8266 <xsd:any namespace="##any" processContents="lax">
8267     <xsd:annotation>
8268         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
8269     </xsd:annotation>
8270 </xsd:any>
8271 </xsd:choice>
8272 </xsd:complexType>
8273 <xsd:complexType name="EventTypeReply">
8274     <xsd:annotation>
8275         <xsd:documentation xml:lang="en"> Event type enumeration object replies. </xsd:documentation>
8276     </xsd:annotation>
8277     <xsd:choice>
8278         <xsd:element name="getSupportedTypes" type="rp:EventTypeListReturnValue"/>
8279     </xsd:choice>
8280 </xsd:complexType>
8281 <xsd:complexType name="FieldNameReply">
8282     <xsd:annotation>
8283         <xsd:documentation xml:lang="en"> Field Name enumeration object replies. </xsd:documentation>
8284     </xsd:annotation>
8285     <xsd:choice>
8286         <xsd:element name="getSupportedNames" type="rp:FieldNameListReturnType"/>
8287     </xsd:choice>
8288 </xsd:complexType>
8289 <xsd:complexType name="NotificationChannelReply">
8290     <xsd:annotation>
8291         <xsd:documentation xml:lang="en"> Notification channel object replies </xsd:documentation>
8292     </xsd:annotation>
8293     <xsd:choice>
8294         <xsd:element name="create" type="rp:NotificationChannelReturnType">
8295             <xsd:annotation>
8296                 <xsd:documentation xml:lang="en"> reply for create Notification channel </xsd:documentation>
8297             </xsd:annotation>
8298         </xsd:element>
8299         <xsd:element name="getName" type="rp:stringReturnType">
8300             <xsd:annotation>
8301                 <xsd:documentation>reply for get name</xsd:documentation>
8302             </xsd:annotation>
8303         </xsd:element>
8304         <xsd:element name="getAddress" type="rp:AddressReturnType">
8305             <xsd:annotation>
8306                 <xsd:documentation xml:lang="en"> reply for get Address</xsd:documentation>
8307             </xsd:annotation>
8308         </xsd:element>
8309         <xsd:element name="getEffectiveAddress" type="rp:AddressReturnType">
8310             <xsd:annotation>
8311                 <xsd:documentation>reply for get effective address</xsd:documentation>
8312             </xsd:annotation>
8313         </xsd:element>
8314         <xsd:element name="setAddress" type="rp:intReturnType">
8315             <xsd:annotation>
8316                 <xsd:documentation xml:lang="en"> reply for set Address</xsd:documentation>

```

```

8317     </xsd:annotation>
8318 </xsd:element>
8319 <xsd:element name="getDataSelector" type="rp:DataSelectorReturnType">
8320     <xsd:annotation>
8321         <xsd:documentation xml:lang="en"> reply for get DataSelector</xsd:documentation>
8322     </xsd:annotation>
8323 </xsd:element>
8324 <xsd:element name="setDataSelector" type="rp:noParamType">
8325     <xsd:annotation>
8326         <xsd:documentation xml:lang="en"> reply for set data selector</xsd:documentation>
8327     </xsd:annotation>
8328 </xsd:element>
8329 <xsd:element name="addSources" type="rp:noParamType">
8330     <xsd:annotation>
8331         <xsd:documentation xml:lang="en"> reply for add Sources</xsd:documentation>
8332     </xsd:annotation>
8333 </xsd:element>
8334 <xsd:element name="removeSources" type="rp:noParamType">
8335     <xsd:annotation>
8336         <xsd:documentation xml:lang="en"> reply for remove Sources</xsd:documentation>
8337     </xsd:annotation>
8338 </xsd:element>
8339 <xsd:element name="removeAllSources" type="rp:noParamType">
8340     <xsd:annotation>
8341         <xsd:documentation xml:lang="en"> reply for remove all Sources</xsd:documentation>
8342     </xsd:annotation>
8343 </xsd:element>
8344 <xsd:element name="getSource" type="rp:SourceReturnType">
8345     <xsd:annotation>
8346         <xsd:documentation xml:lang="en"> reply for get Source</xsd:documentation>
8347     </xsd:annotation>
8348 </xsd:element>
8349 <xsd:element name="getAllSources" type="rp:SourceListReturnType">
8350     <xsd:annotation>
8351         <xsd:documentation xml:lang="en"> reply for get All Sources</xsd:documentation>
8352     </xsd:annotation>
8353 </xsd:element>
8354 <xsd:element name="addNotificationTriggers" type="rp:noParamType">
8355     <xsd:annotation>
8356         <xsd:documentation xml:lang="en"> reply for add notification triggers</xsd:documentation>
8357     </xsd:annotation>
8358 </xsd:element>
8359 <xsd:element name="removeNotificationTriggers" type="rp:noParamType">
8360     <xsd:annotation>
8361         <xsd:documentation xml:lang="en"> reply for remove notification triggers</xsd:documentation>
8362     </xsd:annotation>
8363 </xsd:element>
8364 <xsd:element name="removeAllNotificationTriggers" type="rp:noParamType">
8365     <xsd:annotation>
8366         <xsd:documentation xml:lang="en"> reply for remove all notification triggers</xsd:documentation>
8367     </xsd:annotation>
8368 </xsd:element>
8369 <xsd:element name="getNotificationTrigger" type="rp:TriggerReturnType">
8370     <xsd:annotation>
8371         <xsd:documentation xml:lang="en"> reply for get notification trigger</xsd:documentation>

```

```

8372     </xsd:annotation>
8373 </xsd:element>
8374 <xsd:element name="getAllNotificationTriggers" type="rp:TriggerListReturnType">
8375     <xsd:annotation>
8376         <xsd:documentation xml:lang="en"> reply for get all notification triggers</xsd:documentation>
8377     </xsd:annotation>
8378 </xsd:element>
8379 <xsd:element name="readQueuedData">
8380     <xsd:annotation>
8381         <xsd:documentation xml:lang="en"> reply for read queued data</xsd:documentation>
8382     </xsd:annotation>
8383     <xsd:complexType>
8384         <xsd:sequence>
8385             <xsd:element name="returnValue" type="rp:ReadReportType">
8386                 <xsd:annotation>
8387                     <xsd:documentation xml:lang="en"> Read report of the queued data in
8388                         the channel</xsd:documentation>
8389                 </xsd:annotation>
8390             </xsd:element>
8391         </xsd:sequence>
8392     </xsd:complexType>
8393 </xsd:element>
8394 <xsd:any namespace="##any" processContents="lax">
8395     <xsd:annotation>
8396         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
8397     </xsd:annotation>
8398 </xsd:any>
8399 </xsd:choice>
8400 </xsd:complexType>
8401 <xsd:complexType name="ReaderDeviceReply">
8402     <xsd:choice>
8403         <xsd:annotation>
8404             <xsd:documentation xml:lang="en"> Reader device object replies </xsd:documentation>
8405         </xsd:annotation>
8406         <xsd:element name="getEPC" type="rp:epcReturnType">
8407             <xsd:annotation>
8408                 <xsd:documentation xml:lang="en"> reply for get EPC</xsd:documentation>
8409             </xsd:annotation>
8410         </xsd:element>
8411         <xsd:element name="getManufacturer" type="rp:stringReturnType">
8412             <xsd:annotation>
8413                 <xsd:documentation>reply for getManufacturer</xsd:documentation>
8414             </xsd:annotation>
8415         </xsd:element>
8416         <xsd:element name="getModel" type="rp:stringReturnType">
8417             <xsd:annotation>
8418                 <xsd:documentation>reply for getModel</xsd:documentation>
8419             </xsd:annotation>
8420         </xsd:element>
8421         <xsd:element name="getHandle" type="rp:intReturnType">
8422             <xsd:annotation>
8423                 <xsd:documentation xml:lang="en"> reply for get Handle</xsd:documentation>
8424             </xsd:annotation>
8425         </xsd:element>
8426         <xsd:element name="setHandle" type="rp:noParamType">

```

```

8427     <xsd:annotation>
8428         <xsd:documentation xml:lang="en"> reply for set Handle</xsd:documentation>
8429     </xsd:annotation>
8430 </xsd:element>
8431 <xsd:element name="getName" type="rp:stringReturnType">
8432     <xsd:annotation>
8433         <xsd:documentation>reply for get Name</xsd:documentation>
8434     </xsd:annotation>
8435 </xsd:element>
8436 <xsd:element name="setName" type="rp:noParamType">
8437     <xsd:annotation>
8438         <xsd:documentation xml:lang="en"> reply for set Name</xsd:documentation>
8439     </xsd:annotation>
8440 </xsd:element>
8441 <xsd:element name="getRole" type="rp:stringReturnType">
8442     <xsd:annotation>
8443         <xsd:documentation xml:lang="en"> reply for get Role</xsd:documentation>
8444     </xsd:annotation>
8445 </xsd:element>
8446 <xsd:element name="setRole" type="rp:noParamType">
8447     <xsd:annotation>
8448         <xsd:documentation xml:lang="en"> reply for set Role</xsd:documentation>
8449     </xsd:annotation>
8450 </xsd:element>
8451 <xsd:element name="getTimeTicks" type="rp:intReturnType">
8452     <xsd:annotation>
8453         <xsd:documentation xml:lang="en"> reply for get Time Ticks</xsd:documentation>
8454     </xsd:annotation>
8455 </xsd:element>
8456 <xsd:element name="getTimeUTC" type="rp:timeStampReturnType">
8457     <xsd:annotation>
8458         <xsd:documentation xml:lang="en"> reply for get Time UTC</xsd:documentation>
8459     </xsd:annotation>
8460 </xsd:element>
8461 <xsd:element name="setTimeUTC" type="rp:noParamType">
8462     <xsd:annotation>
8463         <xsd:documentation xml:lang="en"> reply for set Time UTC</xsd:documentation>
8464     </xsd:annotation>
8465 </xsd:element>
8466 <xsd:element name="getManufacturerDescription" type="rp:stringReturnType">
8467     <xsd:annotation>
8468         <xsd:documentation xml:lang="en"> reply for get Manufacturer description</xsd:documentation>
8469     </xsd:annotation>
8470 </xsd:element>
8471 <xsd:element name="getCurrentSource" type="rp:SourceReturnType">
8472     <xsd:annotation>
8473         <xsd:documentation xml:lang="en"> reply for get current source</xsd:documentation>
8474     </xsd:annotation>
8475 </xsd:element>
8476 <xsd:element name="setCurrentSource" type="rp:noParamType">
8477     <xsd:annotation>
8478         <xsd:documentation xml:lang="en"> reply for set current source</xsd:documentation>
8479     </xsd:annotation>
8480 </xsd:element>
8481 <xsd:element name="getCurrentDataSelector" type="rp:DataSelectorReturnType">

```

```

8482     <xsd:annotation>
8483         <xsd:documentation xml:lang="en"> reply for get current data selector</xsd:documentation>
8484     </xsd:annotation>
8485 </xsd:element>
8486 <xsd:element name="setCurrentDataSelector" type="rp:noParamType">
8487     <xsd:annotation>
8488         <xsd:documentation xml:lang="en"> reply for set current data selector</xsd:documentation>
8489     </xsd:annotation>
8490 </xsd:element>
8491 <xsd:element name="removeSources" type="rp:noParamType">
8492     <xsd:annotation>
8493         <xsd:documentation xml:lang="en"> reply for remove sources</xsd:documentation>
8494     </xsd:annotation>
8495 </xsd:element>
8496 <xsd:element name="removeAllSources" type="rp:noParamType">
8497     <xsd:annotation>
8498         <xsd:documentation xml:lang="en">reply for remove all sources</xsd:documentation>
8499     </xsd:annotation>
8500 </xsd:element>
8501 <xsd:element name="getSource" type="rp:SourceReturnType">
8502     <xsd:annotation>
8503         <xsd:documentation xml:lang="en"> reply for get source</xsd:documentation>
8504     </xsd:annotation>
8505 </xsd:element>
8506 <xsd:element name="getAllSources" type="rp:SourceListReturnType">
8507     <xsd:annotation>
8508         <xsd:documentation xml:lang="en"> reply for get all sources</xsd:documentation>
8509     </xsd:annotation>
8510 </xsd:element>
8511 <xsd:element name="removeDataSelectors" type="rp:noParamType">
8512     <xsd:annotation>
8513         <xsd:documentation xml:lang="en"> reply for remove data selectors</xsd:documentation>
8514     </xsd:annotation>
8515 </xsd:element>
8516 <xsd:element name="removeAllDataSelectors" type="rp:noParamType">
8517     <xsd:annotation>
8518         <xsd:documentation xml:lang="en"> reply for remove all data selectors</xsd:documentation>
8519     </xsd:annotation>
8520 </xsd:element>
8521 <xsd:element name="getDataSelector" type="rp:DataSelectorReturnType">
8522     <xsd:annotation>
8523         <xsd:documentation xml:lang="en"> reply for get data selector</xsd:documentation>
8524     </xsd:annotation>
8525 </xsd:element>
8526 <xsd:element name="getAllDataSelectors" type="rp:DataSelectorListReturnType">
8527     <xsd:annotation>
8528         <xsd:documentation xml:lang="en"> reply for get all data selectors</xsd:documentation>
8529     </xsd:annotation>
8530 </xsd:element>
8531 <xsd:element name="removeNotificationChannels" type="rp:noParamType">
8532     <xsd:annotation>
8533         <xsd:documentation xml:lang="en"> reply for remove notification channels</xsd:documentation>
8534     </xsd:annotation>
8535 </xsd:element>
8536 <xsd:element name="removeAllNotificationChannels" type="rp:noParamType">

```

```

8537     <xsd:annotation>
8538         <xsd:documentation>reply for remove all notification channels</xsd:documentation>
8539     </xsd:annotation>
8540 </xsd:element>
8541 <xsd:element name="getNotificationChannel" type="rp:NotificationChannelReturnType">
8542     <xsd:annotation>
8543         <xsd:documentation xml:lang="en"> reply for get notification channel</xsd:documentation>
8544     </xsd:annotation>
8545 </xsd:element>
8546 <xsd:element name="getAllNotificationChannels" type="rp:NotificationChannelListReturnType">
8547     <xsd:annotation>
8548         <xsd:documentation xml:lang="en"> reply for get all notification channels</xsd:documentation>
8549     </xsd:annotation>
8550 </xsd:element>
8551 <xsd:element name="removeTriggers" type="rp:noParamType">
8552     <xsd:annotation>
8553         <xsd:documentation>reply for remove triggers</xsd:documentation>
8554     </xsd:annotation>
8555 </xsd:element>
8556 <xsd:element name="removeAllTriggers" type="rp:noParamType">
8557     <xsd:annotation>
8558         <xsd:documentation>reply to remove all triggers</xsd:documentation>
8559     </xsd:annotation>
8560 </xsd:element>
8561 <xsd:element name="getTrigger" type="rp:TriggerReturnType">
8562     <xsd:annotation>
8563         <xsd:documentation>reply to get trigger</xsd:documentation>
8564     </xsd:annotation>
8565 </xsd:element>
8566 <xsd:element name="getAllTriggers" type="rp:TriggerListReturnType">
8567     <xsd:annotation>
8568         <xsd:documentation>reply to get all triggers</xsd:documentation>
8569     </xsd:annotation>
8570 </xsd:element>
8571 <xsd:element name="removeTagSelectors" type="rp:noParamType">
8572     <xsd:annotation>
8573         <xsd:documentation>reply for remove tag selectors</xsd:documentation>
8574     </xsd:annotation>
8575 </xsd:element>
8576 <xsd:element name="removeAllTagSelectors" type="rp:noParamType">
8577     <xsd:annotation>
8578         <xsd:documentation>reply to remove all tag selectors</xsd:documentation>
8579     </xsd:annotation>
8580 </xsd:element>
8581 <xsd:element name="getTagSelector" type="rp:TagSelectorReturnType">
8582     <xsd:annotation>
8583         <xsd:documentation>reply to get tag selector</xsd:documentation>
8584     </xsd:annotation>
8585 </xsd:element>
8586 <xsd:element name="getAllTagSelectors" type="rp:TagSelectorListReturnType">
8587     <xsd:annotation>
8588         <xsd:documentation>reply to get all tag selectors</xsd:documentation>
8589     </xsd:annotation>
8590 </xsd:element>
8591 <xsd:element name="removeTagFields" type="rp:noParamType">

```

```

8592     <xsd:annotation>
8593         <xsd:documentation>reply to removeTagFields</xsd:documentation>
8594     </xsd:annotation>
8595 </xsd:element>
8596 <xsd:element name="removeAllTagFields" type="rp:noParamType">
8597     <xsd:annotation>
8598         <xsd:documentation>reply to removeAllTagFields</xsd:documentation>
8599     </xsd:annotation>
8600 </xsd:element>
8601 <xsd:element name="getTagField" type="rp:TagFieldReturnType">
8602     <xsd:annotation>
8603         <xsd:documentation>reply to getTagField</xsd:documentation>
8604     </xsd:annotation>
8605 </xsd:element>
8606 <xsd:element name="getAllTagFields" type="rp:TagFieldListReturnType">
8607     <xsd:annotation>
8608         <xsd:documentation>reply to getAllTagFields</xsd:documentation>
8609     </xsd:annotation>
8610 </xsd:element>
8611 <xsd:element name="resetToDefaultSettings" type="rp:noParamType">
8612     <xsd:annotation>
8613         <xsd:documentation>reply to resetToDefaultSettings</xsd:documentation>
8614     </xsd:annotation>
8615 </xsd:element>
8616 <xsd:element name="reboot" type="rp:noParamType">
8617     <xsd:annotation>
8618         <xsd:documentation xml:lang="en">reply for reboot</xsd:documentation>
8619     </xsd:annotation>
8620 </xsd:element>
8621 <xsd:element name="goodbye" type="rp:noParamType">
8622     <xsd:annotation>
8623         <xsd:documentation xml:lang="en">reply for good bye</xsd:documentation>
8624     </xsd:annotation>
8625 </xsd:element>
8626 <xsd:element name="getReadPoint" type="rp:ReadPointReturnType">
8627     <xsd:annotation>
8628         <xsd:documentation>reply to getReadPoint</xsd:documentation>
8629     </xsd:annotation>
8630 </xsd:element>
8631 <xsd:element name="getAllReadPoints" type="rp:ReadPointListReturnType">
8632     <xsd:annotation>
8633         <xsd:documentation xml:lang="en"> reply for get all read points</xsd:documentation>
8634     </xsd:annotation>
8635 </xsd:element>
8636 <xsd:any namespace="##any" processContents="lax">
8637     <xsd:annotation>
8638         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
8639     </xsd:annotation>
8640 </xsd:any>
8641 </xsd:choice>
8642 </xsd:complexType>
8643 <xsd:complexType name="TagSelectorReply">
8644     <xsd:choice>
8645         <xsd:annotation>
8646             <xsd:documentation xml:lang="en"> Tag selector object replies </xsd:documentation>

```

```

8647 </xsd:annotation>
8648 <xsd:element name="create" type="rp:TagSelectorReturnType">
8649   <xsd:annotation>
8650     <xsd:documentation xml:lang="en"> reply for create tag selector</xsd:documentation>
8651   </xsd:annotation>
8652 </xsd:element>
8653 <xsd:element name="getMaxNumberSupported" type="rp:intReturnType">
8654   <xsd:annotation>
8655     <xsd:documentation xml:lang="en"> reply for get max supported tag selectors</xsd:documentation>
8656   </xsd:annotation>
8657 </xsd:element>
8658 <xsd:element name="getName" type="rp:stringReturnType">
8659   <xsd:annotation>
8660     <xsd:documentation>reply for get name</xsd:documentation>
8661   </xsd:annotation>
8662 </xsd:element>
8663 <xsd:element name="getTagField" type="rp:TagFieldReturnType">
8664   <xsd:annotation>
8665     <xsd:documentation>reply to getTagField</xsd:documentation>
8666   </xsd:annotation>
8667 </xsd:element>
8668 <xsd:element name="getValue" type="rp:HexStringReturnType">
8669   <xsd:annotation>
8670     <xsd:documentation xml:lang="en"> reply for get tag selector value</xsd:documentation>
8671   </xsd:annotation>
8672 </xsd:element>
8673 <xsd:element name="getMask" type="rp:HexStringReturnType">
8674   <xsd:annotation>
8675     <xsd:documentation xml:lang="en"> reply for get tag selector mask</xsd:documentation>
8676   </xsd:annotation>
8677 </xsd:element>
8678 <xsd:element name="getInclusiveFlag" type="rp:booleanReturnType">
8679   <xsd:annotation>
8680     <xsd:documentation xml:lang="en"> reply for get Inclusive flag</xsd:documentation>
8681   </xsd:annotation>
8682 </xsd:element>
8683 <xsd:any namespace="##any" processContents="lax">
8684   <xsd:annotation>
8685     <xsd:documentation>For standard and vendor extensions</xsd:documentation>
8686   </xsd:annotation>
8687 </xsd:any>
8688 </xsd:choice>
8689 </xsd:complexType>
8690 <xsd:complexType name="ReadPointReply">
8691   <xsd:choice>
8692     <xsd:element name="getName" type="rp:stringReturnType">
8693       <xsd:annotation>
8694         <xsd:documentation>get the name of the read point</xsd:documentation>
8695       </xsd:annotation>
8696     </xsd:element>
8697     <xsd:any namespace="##any" processContents="lax">
8698       <xsd:annotation>
8699         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
8700       </xsd:annotation>
8701     </xsd:any>

```

```

8702     </xsd:choice>
8703 </xsd:complexType>
8704 <xsd:complexType name="SourceReply">
8705     <xsd:choice>
8706         <xsd:annotation>
8707             <xsd:documentation xml:lang="en"> Source object replies </xsd:documentation>
8708         </xsd:annotation>
8709         <xsd:element name="create" type="rp:SourceReturnType">
8710             <xsd:annotation>
8711                 <xsd:documentation xml:lang="en"> reply for create source</xsd:documentation>
8712             </xsd:annotation>
8713         </xsd:element>
8714         <xsd:element name="getName" type="rp:stringReturnType">
8715             <xsd:annotation>
8716                 <xsd:documentation>reply for get name</xsd:documentation>
8717             </xsd:annotation>
8718         </xsd:element>
8719         <xsd:element name="isFixed" type="rp:booleanReturnType">
8720             <xsd:annotation>
8721                 <xsd:documentation xml:lang="en"> reply for is fixed</xsd:documentation>
8722             </xsd:annotation>
8723         </xsd:element>
8724         <xsd:element name="addReadPoints" type="rp:noParamType">
8725             <xsd:annotation>
8726                 <xsd:documentation xml:lang="en"> reply for add read points</xsd:documentation>
8727             </xsd:annotation>
8728         </xsd:element>
8729         <xsd:element name="removeReadPoints" type="rp:noParamType">
8730             <xsd:annotation>
8731                 <xsd:documentation xml:lang="en"> reply for remove read points</xsd:documentation>
8732             </xsd:annotation>
8733         </xsd:element>
8734         <xsd:element name="removeAllReadPoints" type="rp:noParamType">
8735             <xsd:annotation>
8736                 <xsd:documentation xml:lang="en"> reply for remove all read points</xsd:documentation>
8737             </xsd:annotation>
8738         </xsd:element>
8739         <xsd:element name="getReadPoint" type="rp:ReadPointReturnType">
8740             <xsd:annotation>
8741                 <xsd:documentation xml:lang="en"> reply for get read point</xsd:documentation>
8742             </xsd:annotation>
8743         </xsd:element>
8744         <xsd:element name="getAllReadPoints" type="rp:ReadPointListReturnType">
8745             <xsd:annotation>
8746                 <xsd:documentation xml:lang="en"> reply for get all read points</xsd:documentation>
8747             </xsd:annotation>
8748         </xsd:element>
8749         <xsd:element name="addReadTriggers" type="rp:noParamType">
8750             <xsd:annotation>
8751                 <xsd:documentation xml:lang="en"> reply for add read triggers</xsd:documentation>
8752             </xsd:annotation>
8753         </xsd:element>
8754         <xsd:element name="removeReadTriggers" type="rp:noParamType">
8755             <xsd:annotation>
8756                 <xsd:documentation xml:lang="en"> reply for remove read triggers</xsd:documentation>

```

```

8757     </xsd:annotation>
8758 </xsd:element>
8759 <xsd:element name="removeAllReadTriggers" type="rp:noParamType">
8760     <xsd:annotation>
8761         <xsd:documentation xml:lang="en"> reply for remove all read triggers</xsd:documentation>
8762     </xsd:annotation>
8763 </xsd:element>
8764 <xsd:element name="getReadTrigger" type="rp:TriggerReturnType">
8765     <xsd:annotation>
8766         <xsd:documentation xml:lang="en"> reply for get read trigger</xsd:documentation>
8767     </xsd:annotation>
8768 </xsd:element>
8769 <xsd:element name="getAllReadTriggers" type="rp:TriggerListReturnType">
8770     <xsd:annotation>
8771         <xsd:documentation xml:lang="en"> reply for get all read triggers</xsd:documentation>
8772     </xsd:annotation>
8773 </xsd:element>
8774 <xsd:element name="addTagSelectors" type="rp:noParamType">
8775     <xsd:annotation>
8776         <xsd:documentation xml:lang="en"> reply for add tag selectors</xsd:documentation>
8777     </xsd:annotation>
8778 </xsd:element>
8779 <xsd:element name="removeTagSelectors" type="rp:noParamType">
8780     <xsd:annotation>
8781         <xsd:documentation xml:lang="en"> reply for remove tag selectors</xsd:documentation>
8782     </xsd:annotation>
8783 </xsd:element>
8784 <xsd:element name="removeAllTagSelectors" type="rp:noParamType">
8785     <xsd:annotation>
8786         <xsd:documentation xml:lang="en"> reply for remove tag selectors</xsd:documentation>
8787     </xsd:annotation>
8788 </xsd:element>
8789 <xsd:element name="getTagSelector" type="rp:TagSelectorReturnType">
8790     <xsd:annotation>
8791         <xsd:documentation xml:lang="en"> reply for get tag selector</xsd:documentation>
8792     </xsd:annotation>
8793 </xsd:element>
8794 <xsd:element name="getAllTagSelectors" type="rp:TagSelectorListReturnType">
8795     <xsd:annotation>
8796         <xsd:documentation xml:lang="en"> reply for get all tag selectors</xsd:documentation>
8797     </xsd:annotation>
8798 </xsd:element>
8799 <xsd:element name="getGlimpsedTimeout" type="rp:intReturnType">
8800     <xsd:annotation>
8801         <xsd:documentation xml:lang="en"> reply for get glimpsed timeout</xsd:documentation>
8802     </xsd:annotation>
8803 </xsd:element>
8804 <xsd:element name="setGlimpsedTimeout" type="rp:noParamType">
8805     <xsd:annotation>
8806         <xsd:documentation xml:lang="en"> reply for set glimpsed timeout</xsd:documentation>
8807     </xsd:annotation>
8808 </xsd:element>
8809 <xsd:element name="getObservedThreshold" type="rp:intReturnType">
8810     <xsd:annotation>
8811         <xsd:documentation xml:lang="en"> reply for get observed threshold</xsd:documentation>

```

```

8812     </xsd:annotation>
8813 </xsd:element>
8814 <xsd:element name="setObservedThreshold" type="rp:noParamType">
8815     <xsd:annotation>
8816         <xsd:documentation xml:lang="en"> reply for set observed threshold</xsd:documentation>
8817     </xsd:annotation>
8818 </xsd:element>
8819 <xsd:element name="getObservedTimeout" type="rp:intReturnType">
8820     <xsd:annotation>
8821         <xsd:documentation xml:lang="en"> reply for get observed timeout</xsd:documentation>
8822     </xsd:annotation>
8823 </xsd:element>
8824 <xsd:element name="setObservedTimeout" type="rp:noParamType">
8825     <xsd:annotation>
8826         <xsd:documentation xml:lang="en"> reply for set observed timeout</xsd:documentation>
8827     </xsd:annotation>
8828 </xsd:element>
8829 <xsd:element name="getLostTimeout" type="rp:intReturnType">
8830     <xsd:annotation>
8831         <xsd:documentation xml:lang="en"> reply for get lost timeout</xsd:documentation>
8832     </xsd:annotation>
8833 </xsd:element>
8834 <xsd:element name="setLostTimeout" type="rp:noParamType">
8835     <xsd:annotation>
8836         <xsd:documentation xml:lang="en"> set lost timeout</xsd:documentation>
8837     </xsd:annotation>
8838 </xsd:element>
8839 <xsd:element name="rawReadIDs">
8840     <xsd:annotation>
8841         <xsd:documentation xml:lang="en"> reply for raw read ids</xsd:documentation>
8842     </xsd:annotation>
8843     <xsd:complexType>
8844         <xsd:sequence>
8845             <xsd:element name="returnValue" type="rp:ReadReportType">
8846                 <xsd:annotation>
8847                     <xsd:documentation xml:lang="en"> Read report of ids</xsd:documentation>
8848                 </xsd:annotation>
8849             </xsd:element>
8850         </xsd:sequence>
8851     </xsd:complexType>
8852 </xsd:element>
8853 <xsd:element name="readIDs">
8854     <xsd:annotation>
8855         <xsd:documentation xml:lang="en"> reply for read ids</xsd:documentation>
8856     </xsd:annotation>
8857     <xsd:complexType>
8858         <xsd:sequence>
8859             <xsd:element name="returnValue" type="rp:ReadReportType">
8860                 <xsd:annotation>
8861                     <xsd:documentation xml:lang="en"> Read report of ids</xsd:documentation>
8862                 </xsd:annotation>
8863             </xsd:element>
8864         </xsd:sequence>
8865     </xsd:complexType>
8866 </xsd:element>

```

```

8867 <xsd:element name="read">
8868   <xsd:annotation>
8869     <xsd:documentation>reply for read</xsd:documentation>
8870   </xsd:annotation>
8871   <xsd:complexType>
8872     <xsd:sequence>
8873       <xsd:element name="returnValue" type="rp:ReadReportType">
8874         <xsd:annotation>
8875           <xsd:documentation>Read report of ids</xsd:documentation>
8876         </xsd:annotation>
8877       </xsd:element>
8878     </xsd:sequence>
8879   </xsd:complexType>
8880 </xsd:element>
8881 <xsd:element name="writeID" type="rp:noParamType">
8882   <xsd:annotation>
8883     <xsd:documentation xml:lang="en"> reply for write id</xsd:documentation>
8884   </xsd:annotation>
8885 </xsd:element>
8886 <xsd:element name="write" type="rp:noParamType">
8887   <xsd:annotation>
8888     <xsd:documentation>reply for write</xsd:documentation>
8889   </xsd:annotation>
8890 </xsd:element>
8891 <xsd:element name="kill" type="rp:noParamType">
8892   <xsd:annotation>
8893     <xsd:documentation xml:lang="en"> reply for kill</xsd:documentation>
8894   </xsd:annotation>
8895 </xsd:element>
8896 <xsd:element name="getReadCyclesPerTrigger" type="rp:intReturnType">
8897   <xsd:annotation>
8898     <xsd:documentation xml:lang="en"> reply for get read cycles per trigger </xsd:documentation>
8899   </xsd:annotation>
8900 </xsd:element>
8901 <xsd:element name="setReadCyclesPerTrigger" type="rp:noParamType">
8902   <xsd:annotation>
8903     <xsd:documentation xml:lang="en">reply for set read cycles per trigger </xsd:documentation>
8904   </xsd:annotation>
8905 </xsd:element>
8906 <xsd:element name="getMaxReadDutyCycle" type="rp:intReturnType">
8907   <xsd:annotation>
8908     <xsd:documentation xml:lang="en">reply for get read duty cycle percentage</xsd:documentation>
8909   </xsd:annotation>
8910 </xsd:element>
8911 <xsd:element name="setMaxReadDutyCycle" type="rp:noParamType">
8912   <xsd:annotation>
8913     <xsd:documentation xml:lang="en">reply for set read duty cycle percentage</xsd:documentation>
8914   </xsd:annotation>
8915 </xsd:element>
8916 <xsd:element name="getReadTimeout" type="rp:intReturnType">
8917   <xsd:annotation>
8918     <xsd:documentation xml:lang="en"> reply for get read timeout</xsd:documentation>
8919   </xsd:annotation>
8920 </xsd:element>
8921 <xsd:element name="setReadTimeout" type="rp:noParamType">

```

```

8922     <xsd:annotation>
8923         <xsd:documentation xml:lang="en">reply for set read timeout</xsd:documentation>
8924     </xsd:annotation>
8925 </xsd:element>
8926 <xsd:element name="getSession" type="rp:intReturnType">
8927     <xsd:annotation>
8928         <xsd:documentation>reply for get session</xsd:documentation>
8929     </xsd:annotation>
8930 </xsd:element>
8931 <xsd:element name="setSession" type="rp:noParamType">
8932     <xsd:annotation>
8933         <xsd:documentation>reply for set session</xsd:documentation>
8934     </xsd:annotation>
8935 </xsd:element>
8936 <xsd:any namespace="##any" processContents="lax">
8937     <xsd:annotation>
8938         <xsd:documentation>For standard and vendor extensions</xsd:documentation>
8939     </xsd:annotation>
8940 </xsd:any>
8941 </xsd:choice>
8942 </xsd:complexType>
8943 <xsd:complexType name="TriggerReply">
8944     <xsd:choice>
8945         <xsd:annotation>
8946             <xsd:documentation xml:lang="en"> Trigger object replies </xsd:documentation>
8947         </xsd:annotation>
8948         <xsd:element name="create" type="rp:TriggerReturnType">
8949             <xsd:annotation>
8950                 <xsd:documentation xml:lang="en"> reply for create Trigger</xsd:documentation>
8951             </xsd:annotation>
8952         </xsd:element>
8953         <xsd:element name="getMaxNumberSupported" type="rp:intReturnType">
8954             <xsd:annotation>
8955                 <xsd:documentation>reply for get max number of supported triggers</xsd:documentation>
8956             </xsd:annotation>
8957         </xsd:element>
8958         <xsd:element name="getName" type="rp:stringReturnType">
8959             <xsd:annotation>
8960                 <xsd:documentation>reply for get name</xsd:documentation>
8961             </xsd:annotation>
8962         </xsd:element>
8963         <xsd:element name="getType" type="rp:TriggerTypeReturnType">
8964             <xsd:annotation>
8965                 <xsd:documentation xml:lang="en"> reply for get type</xsd:documentation>
8966             </xsd:annotation>
8967         </xsd:element>
8968         <xsd:element name="getValue" type="rp:TriggerValueReturnType">
8969             <xsd:annotation>
8970                 <xsd:documentation xml:lang="en"> reply for get value</xsd:documentation>
8971             </xsd:annotation>
8972         </xsd:element>
8973         <xsd:element name="fire" type="rp:noParamType">
8974             <xsd:annotation>
8975                 <xsd:documentation xml:lang="en"> reply for fire (trigger)</xsd:documentation>
8976             </xsd:annotation>

```

```

8977     </xsd:element>
8978     <xsd:any namespace="##any" processContents="lax">
8979         <xsd:annotation>
8980             <xsd:documentation>For standard and vendor extensions</xsd:documentation>
8981         </xsd:annotation>
8982     </xsd:any>
8983 </xsd:choice>
8984 </xsd:complexType>
8985 <xsd:complexType name="TriggerTypeReply">
8986     <xsd:annotation>
8987         <xsd:documentation xml:lang="en"> Trigger type enumeration object replies. </xsd:documentation>
8988     </xsd:annotation>
8989     <xsd:choice>
8990         <xsd:element name="getSupportedTypes" type="rp:TriggerTypeListReturnType"/>
8991     </xsd:choice>
8992 </xsd:complexType>
8993 <xsd:complexType name="TagFieldReply">
8994     <xsd:choice>
8995         <xsd:element name="create" type="rp:TagFieldReturnType"/>
8996         <xsd:element name="getName" type="rp:stringReturnType"/>
8997         <xsd:element name="getTagFieldName" type="rp:stringReturnType"/>
8998         <xsd:element name="setTagFieldName" type="rp:noParamType"/>
8999         <xsd:element name="getMemoryBank" type="rp:intReturnType"/>
9000         <xsd:element name="setMemoryBank" type="rp:noParamType"/>
9001         <xsd:element name="getOffset" type="rp:intReturnType"/>
9002         <xsd:element name="setOffset" type="rp:noParamType"/>
9003         <xsd:element name="getLength" type="rp:intReturnType"/>
9004         <xsd:element name="setLength" type="rp:noParamType"/>
9005     </xsd:choice>
9006 </xsd:complexType>
9007 </xsd:schema>

```

A.4 Notification Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="urn:epcglobal:rp:xsd:1" xmlns:rp="urn:epcglobal:rp:xsd:1"
xmlns:epcglobal="urn:epcglobal:xsd:1" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="unqualified" attributeFormDefault="unqualified" version="1.0">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      <epcglobal:copyright>Copyright ©2003-2006 Epcglobal Inc., All Rights Reserved.</epcglobal:copyright>
      <epcglobal:disclaimer>EPCglobal Inc., its members, officers, directors, employees, or
        agents shall not be liable for any injury, loss, damages, financial or otherwise,
        arising from, related to, or caused by the use of this document. The use of said
        document shall constitute your express consent to the foregoing exculpation.</epcglobal:disclaimer>
      <epcglobal:specification>Reader Protocol (RP) version 1.1</epcglobal:specification>
    </xsd:documentation>
  </xsd:annotation>
  <xsd:include schemaLocation="RpCommon.xsd"/>
  <xsd:import namespace="urn:epcglobal:xsd:1" schemaLocation="./EpcGlobal.xsd"/>
  <!-- Reader Protocol Notification -->
  <xsd:element name="notification">
    <xsd:annotation>
      <xsd:documentation xml:lang="en"> This element defines a reader protocol notification.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="id" type="xsd:string">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">id of this notification</xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="reader" type="rp:ReaderType" minOccurs="0">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">Reader sending the notification</xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="notifyTriggerName" type="xsd:string" minOccurs="0">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">Trigger causing the notification</xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="notifyChannelName" type="xsd:string" minOccurs="0">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">notification channel</xsd:documentation>
          </xsd:annotation>
        </xsd:element>
        <xsd:choice>
          <xsd:element name="goodBye" type="rp:noParamType">
            <xsd:annotation>
              <xsd:documentation xml:lang="en">sent by reader to gracefully
                close this notification channel</xsd:documentation>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="readReport" type="rp:ReadReportType" maxOccurs="unbounded">
            <xsd:annotation>
              <xsd:documentation xml:lang="en">Read reports</xsd:documentation>
            </xsd:annotation>
          </xsd:element>
        </xsd:choice>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

9062         </xsd:annotation>
9063     </xsd:element>
9064     <xsd:any namespace="##any" processContents="lax" maxOccurs="unbounded">
9065         <xsd:annotation>
9066             <xsd:documentation>A standard or a vendor extension</xsd:documentation>
9067         </xsd:annotation>
9068     </xsd:any>
9069     <!-- Reader Protocol notification types -->
9070     <!-- elements inside are ordered in alphabetical order -->
9071     </xsd:choice>
9072 </xsd:sequence>
9073 </xsd:complexType>
9074 </xsd:element>
9075 <xsd:element name="ackNack">
9076     <xsd:annotation>
9077         <xsd:documentation xml:lang="en"> This element defines a reader protocol notification [reply] ACK/NACK. It is
9078         used when ACK/NACK is enabled for a notification channel and underlying transport doesn't support
9079         ACK.</xsd:documentation>
9080     </xsd:annotation>
9081     <xsd:complexType>
9082         <xsd:complexContent>
9083             <xsd:extension base="epcglobal:Document">
9084                 <xsd:sequence>
9085                     <xsd:element name="id" type="xsd:string">
9086                         <xsd:annotation>
9087                             <xsd:documentation xml:lang="en">id of the notification being
9088         acknowledged/rejected</xsd:documentation>
9089                         </xsd:annotation>
9090                     </xsd:element>
9091                     <xsd:element name="reader" type="rp:ReaderType" minOccurs="0">
9092                         <xsd:annotation>
9093                             <xsd:documentation xml:lang="en">Reader sending the notification</xsd:documentation>
9094                         </xsd:annotation>
9095                     </xsd:element>
9096                     <xsd:element name="notifyChannelName" type="xsd:string" minOccurs="0">
9097                         <xsd:annotation>
9098                             <xsd:documentation xml:lang="en">notification channel</xsd:documentation>
9099                         </xsd:annotation>
9100                     </xsd:element>
9101                     <xsd:element name="isAck" type="xsd:boolean" default="true">
9102                         <xsd:annotation>
9103                             <xsd:documentation xml:lang="en">Whether this ackNack is an Ack.</xsd:documentation>
9104                         </xsd:annotation>
9105                     </xsd:element>
9106                 </xsd:sequence>
9107             </xsd:extension>
9108         </xsd:complexContent>
9109     </xsd:complexType>
9110 </xsd:element>
9111 <xsd:complexType name="ReaderType">
9112     <xsd:annotation>
9113         <xsd:documentation xml:lang="en">Reader Information</xsd:documentation>
9114     </xsd:annotation>
9115     <xsd:sequence>
9116         <xsd:element name="readerEPC" type="epcglobal:EPC" minOccurs="0">

```

```

9117     <xsd:annotation>
9118         <xsd:documentation xml:lang="en">Reader EPC</xsd:documentation>
9119     </xsd:annotation>
9120 </xsd:element>
9121 <xsd:element name="readerName" type="xsd:string" minOccurs="0">
9122     <xsd:annotation>
9123         <xsd:documentation xml:lang="en">Reader name</xsd:documentation>
9124     </xsd:annotation>
9125 </xsd:element>
9126 <xsd:element name="readerHandle" type="xsd:int" minOccurs="0">
9127     <xsd:annotation>
9128         <xsd:documentation xml:lang="en">Reader handle</xsd:documentation>
9129     </xsd:annotation>
9130 </xsd:element>
9131 <xsd:element name="readerRole" type="xsd:string" minOccurs="0">
9132     <xsd:annotation>
9133         <xsd:documentation xml:lang="en">Reader role</xsd:documentation>
9134     </xsd:annotation>
9135 </xsd:element>
9136 <xsd:element name="readerTime" type="rp:ReaderTimeType" minOccurs="0">
9137     <xsd:annotation>
9138         <xsd:documentation xml:lang="en">Reader time</xsd:documentation>
9139     </xsd:annotation>
9140 </xsd:element>
9141 <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
9142 </xsd:sequence>
9143 </xsd:complexType>
9144 </xsd:schema>

```

B References

Normative References

- [ISO 10646-1:2000 Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane
<http://www.iso.org/iso/en/ISOOnline.frontpage>
- [RFC1738] T. Berners-Lee, L. Masinter, M. McCahill, "Uniform Resource Locators (URL)," RFC 1738, December 1994, <http://www.ietf.org/rfc/rfc1738.txt>.
- [RFC2234] D. Crocker, P. Overell, "Augmented BNF for Syntax Specifications: ABNF," RFC 2234, November 1997, <http://www.ietf.org/rfc/rfc2234.txt>.
- [RFC2246] T. Dierks, C. Allen, "The TLS Protocol, Version 1.0," RFC 2246, January 1999, <http://www.ietf.org/rfc/rfc2246.txt>.
- [RFC2396] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," RFC 2396, August 1998, <http://www.ietf.org/rfc/rfc2396.txt>.
- [RFC2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1" RFC 2616, June 1999, <http://www.ietf.org/rfc/rfc2616.txt>
- [RFC2818] E. Rescorla, "HTTP Over TLS" RFC 2818, May 2000, <http://www.ietf.org/rfc/rfc2818.txt>.
- [RFC3268] P. Chown, "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)" RFC 3268, June 2002, <http://www.ietf.org/rfc/rfc3268.txt>
- [RFC3629] F. Yergeau, "UTF-8, a transformation format of ISO 10646" RFC 3629, November 2003, <http://www.ietf.org/rfc/rfc3629.txt>.
- [TDS1.1] EPCglobal, "EPC Tag Data Standards Version 1.1 Rev.1.24," EPCglobal Standard Specification, April 2004, http://www.epcglobalinc.org/standards_technology/EPCTagDataSpecification11rev124.pdf.
- [XML1.0] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, "Extensible Markup Language (XML) 1.0 (Third Edition)," W3C Recommendation, February 2004, <http://www.w3.org/TR/2004/REC-xml-20040204/>.
- [XSD1] H. Thompson, D. Beech, M. Maloney, N. Mendelsohn, "XML Schema Part 1: Structures Second Edition," W3C Recommendation, October 2004, <http://www.w3.org/TR/xmlschema-1/>.
- [XSD2] P. Biron, A. Malhotra, "XML Schema Part 2: Datatypes Second Edition," W3C Recommendation, October 2004, <http://www.w3.org/TR/xmlschema-2/>.

9181 **Informative References**

- 9182 [Gen2 1.0.9] EPCglobal, “Class 1 Generation 2 UHF Air Interface Protocol Standard
9183 Version 1.0.9”, EPCglobal Standard Specification, January 2005,
9184 http://www.epcglobalinc.org/standards_technology/EPCglobal2UHF RFIDProtocolV1091
9185 [22005.pdf](http://www.epcglobalinc.org/standards_technology/EPCglobal2UHF RFIDProtocolV1091)
- 9186 [HF1] “13.56 MHz ISM Band Class 1 Radio Frequency Identification Tag Interface
9187 Specification: Candidate Recommendation, Version 1.0.0,” Auto-ID Center Technical
9188 Report, <http://develop.autoidcenter.org/TR/HF-class1.pdf>.
- 9189 [ISODir2] ISO, “Rules for the structure and drafting of International Standards
9190 (ISO/IEC Directives, Part 2, 2001, 4th edition),” July 2002.
- 9191 [UHF0] “Draft protocol specification for a 900 MHz Class 0 Radio Frequency
9192 Identification Tag,” Auto-ID Center Technical Report,
9193 <http://develop.autoidcenter.org/TR/UHF-class0.pdf>.
- 9194 [UHF1] “860MHz–930MHz Class I Radio Frequency Identification Tag Radio
9195 Frequency & Logical Communication Interface Specification Candidate
9196 Recommendation, Version 1.0.1,” Auto-ID Center Technical Report,
9197 <http://develop.autoidcenter.org/TR/UHF-class1.pdf>.
- 9198