



Powered by GS1

The Application Level Events (ALE) Specification, Version 1.0

EPCglobal Ratified Specification
Version of September 15, 2005

Abstract

This document specifies an interface through which clients may obtain filtered, consolidated Electronic Product Code™ (EPC) data from a variety of sources. The design of this interface recognizes that in most EPC processing systems, there is a level of processing that reduces the volume of data that comes directly from EPC data sources such as RFID readers into coarser “events” of interest to applications. It also recognizes that decoupling these applications from the physical layers of infrastructure offers cost and flexibility advantages to technology providers and end-users alike.

The processing done at this layer typically involves: (1) *receiving* EPCs from one or more data sources such as readers; (2) *accumulating* data over intervals of time, *filtering* to eliminate duplicate EPCs and EPCs that are not of interest, and *counting* and *grouping* EPCs to reduce the volume of data; and (3) *reporting* in various forms. The interface described herein, and the functionality it implies, is called “Application Level Events,” or ALE.

The role of the ALE interface within the EPCglobal Network™ Architecture is to provide independence between the infrastructure components that acquire the raw EPC data, the architectural component(s) that filter & count that data, and the applications that use the data. This allows changes in one without requiring changes in the other, offering significant benefits to both the technology provider and the end-user. The ALE interface described in the present specification achieves this independence through three means:

- It provides a means for clients to specify, in a high-level, declarative way, what EPC data they are interested in, without dictating an implementation. The interface is designed to give implementations the widest possible latitude in selecting strategies for carrying out client requests; such strategies may be influenced by performance goals, the native abilities of readers which may carry out certain filtering or counting operations at the level of firmware or RF protocol, and so forth.
- It provides a standardized format for reporting accumulated, filtered EPC data that is largely independent of where the EPC data originated or how it was processed.
- It abstracts the sources of EPC data into a higher-level notion of “logical reader,” often synonymous with “location,” hiding from clients the details of exactly what physical devices were used to gather EPC data relevant to a particular logical location. This allows changes to occur at the physical layer (for example, replacing a 2-port multi-antenna reader at a loading dock door with three “smart antenna” readers) without affecting client applications. Similarly, it abstracts away the fine-grained details of how data is gathered (*e.g.*, how many individual tag read attempts were carried out). These features of abstraction are a consequence of the way the data specification and reporting aspects of the interface are designed.

The specification includes a formal processing model, an application programming interface (API) described abstractly via UML, and bindings of the API to a WS-i compliant SOAP protocol with associated bindings of the key data types to XML schema.

52 Implementors may provide other bindings, as well as extensions, as provided by the
53 framework of the specification.

54 **Audience for this document**

55 The target audience for this specification includes:

- 56 • EPC Middleware vendors
- 57 • Reader vendors
- 58 • Application developers
- 59 • System integrators

60 **Status of this document**

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

64 This version of the specification was ratified by the EPCglobal Board of Governors on
65 September 23, 2005. It was reviewed and approved by the EPCglobal Business Steering
66 Committee on 14 February 2005 and by the Technical Steering Committee on 2 February
67 2005.

68 Comments on this document should be sent to the EPCglobal Software Action Group
69 Filtering and Collection Working Group mailing list
70 sag_fc@epclinklist.epcglobalinc.org.

71 **Table of Contents**

72	1	Introduction	6
73	2	Role Within the EPCglobal Network Architecture	7
74	3	Terminology and Typographical Conventions.....	9
75	4	ALE Formal Model	9
76	5	Group Reports	13
77	6	Read Cycle Timing.....	13
78	7	Logical Reader Names	14
79	8	ALE API.....	16
80	8.1	ALE – Main API Class	18
81	8.1.1	Error Conditions.....	20
82	8.2	ECSpec	22

83	8.2.1	ECBoundarySpec	23
84	8.2.2	ECTime	25
85	8.2.3	ECTimeUnit	25
86	8.2.4	ECTrigger	25
87	8.2.5	ECReportSpec	25
88	8.2.6	ECReportSetSpec	27
89	8.2.7	ECFilterSpec	27
90	8.2.8	EPC Patterns (non-normative)	27
91	8.2.9	ECGroupSpec	28
92	8.2.10	ECReportOutputSpec	31
93	8.2.11	Validation of ECSpecs	32
94	8.3	ECReports	33
95	8.3.1	ECTerminationCondition	34
96	8.3.2	ECReport	34
97	8.3.3	ECReportGroup	35
98	8.3.4	ECReportGroupList	35
99	8.3.5	ECReportGroupListMember	36
100	8.3.6	ECReportGroupCount	37
101	9	Standard Notification URIs	37
102	9.1	HTTP Notification URI	37
103	9.2	TCP Notification URI	38
104	9.3	FILE Notification URI	38
105	10	XML Schema for Event Cycle Specs and Reports	39
106	10.1	Extensibility Mechanism	39
107	10.2	Schema	42
108	10.3	ECSpec – Example (non-normative)	49
109	10.4	ECReports – Example (non-normative)	49
110	11	SOAP Binding for ALE API	50
111	11.1	SOAP Binding	50
112	12	Use Cases (non-normative)	61
113	13	ALE Scenarios (non-normative)	63
114	13.1	ALE Context	63

115	13.2	Scenarios	64
116	13.2.1	Scenario 1a: Direct Subscription	65
117	13.2.1.1	Assumptions.....	65
118	13.2.1.2	Description.....	66
119	13.2.2	Scenario 1b: Indirect Subscription	66
120	13.2.2.1	Assumptions.....	67
121	13.2.2.2	Description.....	67
122	13.2.3	Scenario 2, 3: Poll, Immediate.....	68
123	13.2.3.1	Assumptions.....	68
124	13.2.3.2	Description.....	69
125	14	Glossary (non-normative)	69
126	15	References.....	70
127			
128			

1 Introduction

This document specifies an interface through which clients may obtain filtered, consolidated EPC data from a variety of sources. The design of this interface recognizes that in most EPC processing systems, there is a level of processing that reduces the volume of data that comes directly from EPC data sources such as RFID readers into coarser “events” of interest to applications. It also recognizes that decoupling these applications from the physical layers of infrastructure offers cost and flexibility advantages to technology providers and end-users alike.

The processing done at this layer typically involves: (1) *receiving* EPCs from one or more data sources such as readers; (2) *accumulating* data over intervals of time, *filtering* to eliminate duplicate EPCs and EPCs that are not of interest, and *counting* and *grouping* EPCs to reduce the volume of data; and (3) *reporting* in various forms. The interface described herein, and the functionality it implies, is called “Application Level Events,” or ALE.

In early versions of the EPCglobal Network Architecture, originating at the Auto-ID Center at the Massachusetts Institute of Technology (MIT), these functions were understood to be part of a specific component termed “Savant.” The term “Savant” has been variously used to refer generically to any software situated between RFID readers and enterprise applications, or more specifically to a particular design for such software as described by an MIT Auto-ID Center document “The Savant Specification Version 0.1” [Savant0.1] or to a later effort by the Auto-ID Center Software Action Group [Savant1.0] that outlined a generalized container framework for such software. Owing to the confusion surrounding the term, the word “Savant” has been deprecated by EPCglobal in favor of more definite specifications of particular functionality. The interface described herein is the first such definite specification.

The role of the ALE interface within the EPCglobal Network Architecture is to provide independence between the infrastructure components that acquire the raw EPC data, the architectural component(s) that filter & count that data, and the applications that use the data. This allows changes in one without requiring changes in the other, offering significant benefits to both the technology provider and the end-user. The ALE interface described in the present specification achieves this independence through three means:

- It provides a means for clients to specify, in a high-level, declarative way, what EPC data they are interested in, without dictating an implementation. The interface is designed to give implementations the widest possible latitude in selecting strategies for carrying out client requests; such strategies may be influenced by performance goals, the native abilities of readers which may carry out certain filtering or counting operations at the level of firmware or RF protocol, and so forth.
- It provides a standardized format for reporting accumulated, filtered EPC data that is largely independent of where the EPC data originated or how it was processed.
- It abstracts the sources of EPC data into a higher-level notion of “logical reader,” often synonymous with “location,” hiding from clients the details of exactly what physical devices were used to gather EPC data relevant to a particular logical

location. This allows changes to occur at the physical layer (for example, replacing a 2-port multi-antenna reader at a loading dock door with three “smart antenna” readers) without affecting client applications. Similarly, it abstracts away the fine-grained details of how data is gathered (*e.g.*, how many individual tag read attempts were carried out). These features of abstraction are a consequence of the way the data specification and reporting aspects of the interface are designed.

Unlike the earlier MIT “Savant Version 0.1” effort, the present specification does *not* specify a particular implementation strategy, or internal interfaces within a specific body of software. Instead, this specification focuses exclusively on one external interface, admitting a wide variety of possible implementations so long as they fulfill the contract of the interface. For example, it is possible to envision an implementation of this interface as an independent piece of software that speaks to RFID readers using their network wireline protocols. It is equally possible, however, to envision another implementation in which the software implementing the interface is part of the reader device itself.

2 Role Within the EPCglobal Network Architecture

EPC technology, especially when implemented using RFID, generates a very large number of object reads throughout the supply chain and eventually into consumer usage. Many of those reads represent non-actionable “noise.” To balance the cost and performance of this with the need for clear accountability and interoperability of the various parts, the design of the EPCglobal Network Architecture seeks to:

1. Drive as much filtering and counting of reads as low in the architecture as possible (*i.e.*, in first preference to readers, then to “middleware”, and as a last resort to “applications”), while meeting application and cost needs;
2. At the same time, minimize the amount of “business logic” embedded in the Tags, Readers and middleware, where business logic is either data or processing logic that is particular to an individual product, product category, industry or business process.

The Application Level Events (ALE) interface specified herein is intended to facilitate these objectives by providing a flexible interface to a standard set of accumulation, filtering, and counting operations that produce “reports” in response to client “requests.” The client will be responsible for interpreting and acting on the meaning of the report (*i.e.*, the “business logic”). The client of the ALE interface may be a traditional “enterprise application,” or it may be new software designed expressly to carry out an EPC-enabled business process but which operates at a higher level than the “middleware” that implements the ALE interface. Hence, the term “Application Level Events” should not be misconstrued to mean that the client of the ALE interface is necessarily a traditional “enterprise application.”

The ALE interface revolves around client requests and the corresponding reports that are produced. Requests can either be: (1) *immediate*, in which information is reported on a one-time basis at the time of the request; or (2) *recurring*, in which information is reported repeatedly whenever an event is detected or at a specified time interval. The

results reported in response to a request can be directed back to the requesting client or to a “third party” specified by the requestor.

This reporting API can be viewed as the interface to a layer of functionality that sits between raw EPC detection events (RFID tag reads or otherwise) and application business logic. We refer to this layer as the Application Level Event (ALE) layer. Note that this document does not specify where ALE-level processing takes place: it could take place within independent software “middleware,” within a suitably capable reader, or some combination, though always with the ALE interface serving as a point of interface to the client. Even when implemented as software middleware, the filtering, counting, and other processing requested by a client may be carried out within the software, or pushed into the readers or other devices. This aspect of the ALE specification is intended explicitly to give freedom to implementers, and to provide a way to take full advantage of a range of reader capabilities (while at the same time avoiding clients from needing to understand the details of those capabilities).

**Application
Business Logic**

**Application Level
Event (ALE) Layer**

**Raw Tag Read
Layer**

In many cases, the client of ALE will be software that incorporates the EPC Information Service (EPCIS), or other business processing software. Since EPCIS is another component of the EPCglobal Network Architecture that deals with higher-level EPC events, it is helpful to understand how ALE differs from EPCIS and other software at higher levels of the architecture. The principal differences are:

- The ALE interface is exclusively oriented towards real-time processing of EPC data, with no persistent storage of EPC data required by the interface (though implementations may employ persistent storage to provide resilience to failures). Business applications, in contrast, typically deal explicitly with historical data and hence are inherently persistent in nature.
- The events communicated through the ALE interface are pure statements of “what, where, and when,” with no business semantics expressed. Business applications, and typically EPCIS-level data, does embed business semantics at some level. For example, at the ALE level, there might be an event that says “at location L, in the time interval T1–T2, the following 100 case-level EPCs and one pallet-level EPC were read.” Within a business application, the corresponding statement might be “at location L, at time T2, it was confirmed that the following 100 cases were aggregated onto the following pallet.” The business-level event, while containing essentially the same EPC data as the ALE event, is at a semantically higher level because it incorporates an understanding of the business process in which the EPC data were obtained.

The distinction between the ALE and EPCIS/business layers is useful because it separates concerns. The ALE layer is concerned with dealing with the mechanics of data gathering, and of filtering down to meaningful events that are a suitable starting point for interpretation by business logic. Business layers are concerned with business process,

and recording events that can serve as the basis for a wide variety of enterprise-level information processing tasks. Within this general framework, there is room for many different approaches to designing systems to meet particular business goals, and it is expected that there will not necessarily be one “right” way to construct systems. Thus, the focus in this specification is not on a particular system architecture, but on creating a very well defined interface that will be useful within a variety of designs.

- A reference to the EPCglobal Network Architecture document should be inserted when EPCglobal publishes such a document.

3 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, with the exception of Section 1 and Section 2, are normative, except where 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, and XML identifiers, 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.

4 ALE Formal Model

Within this specification, the term “Reader” is used to refer to a source of raw EPC data events. An extremely common type of source, of course, is an actual RFID reader, which generates EPC data by using an RF protocol to read EPC codes from RFID tags. But a Reader could just as easily be an EPC-compatible bar code reader, or even a person typing on a keyboard. Moreover, Readers as used in this specification may not necessarily be in one-to-one correspondence with hardware devices; this is explored in more depth in Section 7. Hence, the term “Reader” is just a convenient shorthand for “raw EPC data event source.” When used in this special sense, the word Reader will always be capitalized. For purposes of discussion, it will sometimes be necessary to speak of tags moving within the detection zone of a Reader; while this terminology is directly germane to RFID readers, it should be obvious what the corresponding meaning would be for other types of Readers.

A *read cycle* is the smallest unit of interaction with a Reader. The result of a read cycle is a set of EPCs. In the case of an RFID reader antenna, the EPCs in a read cycle are sometimes those obtained in a single operation of the reader’s RF protocol, though this is

not necessarily the case. The output of a read cycle is the input to the ALE layer; *i.e.*, it is the interface between the Raw Tag Read Layer and the ALE Layer in the diagram of Section 2. As was noted earlier, this interface could be an actual software or network interface between a reader device and a middleware implementation, but this is not necessarily the case. From the ALE perspective, a read cycle is a single event containing a set of EPCs, with nothing more implied.

An *event cycle* is one or more read cycles, from one or more Readers that are to be treated as a unit from a client perspective. It is the smallest unit of interaction between the ALE interface and a client. Referring to the diagram of Section 2, clients in the Application Business Logic Layer specify the boundaries of event cycles to the ALE layer as part of a request for a report.

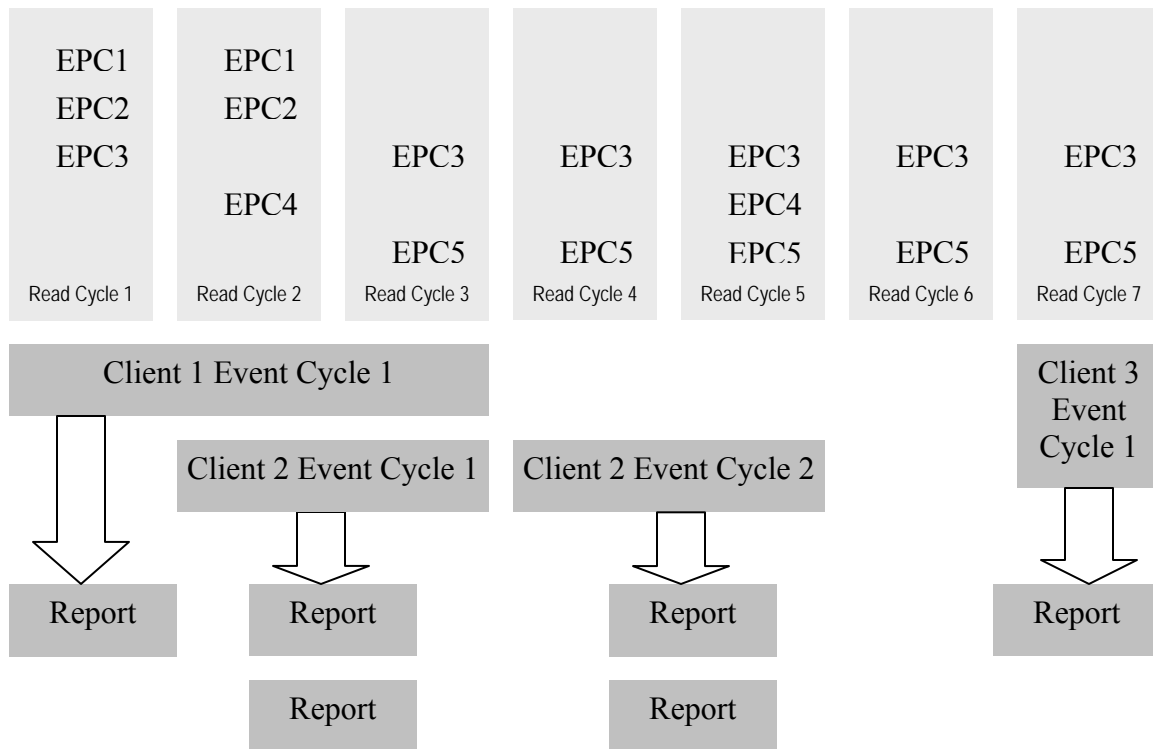
A *report* is data about an event cycle communicated from the ALE implementation to a client. The report is the output of the ALE layer, communicated to the Application Business Logic Layer.

As tags or other carriers of EPC data move in and out of the detection zone of a Reader, the EPCs reported in each read cycle change. Within an event cycle, the same tag may be read several times (if the tag remains within the detection zone of any of the Readers specified for that event cycle). Section 8.2.1 specifies how event cycle boundaries may:

- Extend for a specified duration (interval of real time); *e.g.*, accumulate reads into five-second intervals.
- Occur periodically; *e.g.*, report only every 30 minutes, regardless of the read cycle.
- Be triggered by external events; *e.g.*, an event cycle starts when a pallet on a conveyer triggers an electric eye upstream of a portal, and ends when it crosses a second electric eye downstream of a portal.
- Be delimited when no new EPCs are detected by any Reader specified for that event cycle for a specified interval of time.
- Simply be every read cycle. (This possibility is not provided for in Section 8.2, but may be available through vendor extensions.)

A client must specify one of these methods when requesting a report. (The complete set of available options is described normatively in Section 8.2.1.)

The net picture looks something like this:



326

327 While the diagram shows read cycles arising from a single Reader, in practice a given
 328 event cycle may collect read cycles from more than one Reader. As the diagram
 329 suggests, there may be more than one active event cycle at any point in time. Multiple
 330 active event cycles may start and end with different read cycles, and may overlap in
 331 arbitrary ways. They may gather data from the same Readers, from different Readers, or
 332 from arbitrarily overlapping sets of Readers. Multiple active event cycles could arise
 333 from one client making several simultaneous requests, or from independent clients. In all
 334 cases, however, the same read cycles are shared by all active event cycles that request
 335 data from a given Reader.

336 The set of EPCs in a given read cycle from a given Reader is denoted by S . In the picture
 337 above, $S_1 = \{EPC1, EPC2, EPC3\}$ and $S_2 = \{EPC1, EPC2, EPC4\}$.

338 An event cycle is treated as a unit by clients, so clients do not see any of the internal
 339 structure of the event cycle. All that is relevant, therefore, is the complete set of EPCs
 340 occurring in any of the read cycles that make up the event cycle, from any of the Readers
 341 in the set specified for the event cycle, with duplicates removed. This is simply the union
 342 of the read cycle sets: $E = S_1 \cup S_2 \cup \dots$. In the example above for Client 1 Event
 343 Cycle 1 we have $E_{1.1} = \{EPC1, EPC2, EPC3, EPC4, EPC5\}$.

344 Clients get information about event cycles through reports. A report is specified by a
 345 combination of these three parameters:

- 346 • What set R to report, which may be
- 347 • The *complete* set from the current event cycle $R = E_{cur}$; or

348 • The *differential* set that only includes differences of the current event cycle
349 relative to the previous one (assuming the same event cycle boundaries). This can
350 be the set of additions $R = (E_{cur} - E_{prev})$ or the set of deletions $R = (E_{prev} -$
351 $E_{cur})$, where ‘ $-$ ’ denotes the set difference operator.

352 • An optional filter $F(R)$ to apply, including as part of the standard ALE interface:

353 • One or more object types derived from the “filter bits” of the EPC Tag Data
354 Standard [TDS1.1], including “product” objects (*e.g.*, pallet, case, *etc.*) as well as
355 “location” objects (*e.g.*, warehouse slots, trucks, retail shelves, *etc.*, that contain
356 embedded EPC tags)

357 • A specific list of EPCs

358 • A range of EPCs

359 • Whether to report

360 • The members of the set, $F(R)$ (*i.e.*, the EPCs themselves), possibly grouped as
361 described in Section 5, and in what format (*e.g.*, pure identity URI, tag URI, raw
362 binary, *etc.*);

363 • The quantity, or cardinality, of the set $|F(R)|$, or of the groups making up the set as
364 described in Section 5.

365 The available options are described normatively in Section 8.2.

366 A client may require more than one report from a given event cycle; *e.g.*, a smart shelf
367 application may require both an additions report and a deletions report.

368 This all adds up to an ALE Layer API in which the primary interaction involves: (1) a
369 client specifying: (a) one or more Readers (this is done indirectly, as explained in
370 Section 7) (b) event cycle boundaries as enumerated above, and (c) a set of reports as
371 defined above; and (2) the ALE Layer responding by returning the information implied
372 by that report specification for one or more event cycles. This may be done in a “pull”
373 mode, where the client asks for a report or reports (also specifying how the event cycle is
374 to be delimited) and the ALE Layer in turn initiates or waits for read events, filters/counts
375 the data, and returns the report(s). It may also be done in a “push” mode, where the client
376 registers a subscription with a report set and event cycle boundary specification, and
377 thereafter the ALE Layer asynchronously sends reports to the client when event cycles
378 complete. The complete details of the API, the information required to specify an event
379 cycle, and the information returned to the client when an event cycle completes are
380 spelled out in Sections 8.1, 8.2, and 8.3, respectively. Examples of an event cycle
381 specification and event cycle reports in XML are given in Section 10.

382 Note that because the filtering operations commute with the set union and difference
383 operations, there is a great deal of freedom in how an ALE implementation actually
384 carries out the task of fulfilling a report request. For example, in one implementation,
385 there may be a Reader that is capable of doing filtering directly within the Reader, while
386 in a second implementation the Reader may not be capable of filtering and so software
387 implementing the ALE API must do it. But the ALE API itself need not change – the

388 client specifies the reports, and the implementation of the API decides where best to carry
389 out the requested filtering.

390 **5 Group Reports**

391 Sometimes it is useful to group EPCs read during an event cycle based on portions of the
392 EPC or attributes of the objects identified by the EPCs. For example, in a shipment
393 receipt verification application, it is useful to know the quantity of each type of case (*i.e.*,
394 each distinct case GTIN), but not necessarily the serial number of each case. This
395 requires slightly more complex processing, based on the notion of a grouping operator.

396 A *grouping operator* is a function G that maps an EPC code into some sort of group
397 code g . For example, a grouping operator might map an EPC code into a GTIN group, or
398 simply into the upper bits (manufacturer and product) of the EPC. Other grouping
399 operators might be based on other information available on an EPC tag, such as the filter
400 code that implies the type of object (*i.e.*, pallet, case, item, *etc.*).

401 The notation $S \downarrow g$ means the subset of EPCs $s1, s2, \dots$ in the set S that belong to group g .
402 That is, $S \downarrow g \equiv \{ s \text{ in } S \mid G(s) = g \}$.

403 A *group membership report* for grouping operator G is a set of pairs, where the first
404 element in each pair is a group name g , and the second element is the list of EPCs that
405 fall into that group, *i.e.*, $S \downarrow g$.

406 A *group cardinality report* is similar, but instead of enumerating the EPCs in each group,
407 the group cardinality report just reports how many of each there are. That is, the group
408 cardinality report for grouping operator G is a set of pairs, where the first element in each
409 pair is a group name g , and the second element is the number of EPCs that fall into that
410 group, *i.e.*, $|S \downarrow g|$.

411 Formally, then, the reporting options from the last section are:

- 412 • Whether to report
- 413 • A group membership (group list) report for one or more specified grouping
414 operators G_i , which may include, and may possibly be limited to, the default
415 (unnamed) group. In mathematical notation: $\{ (g, F(R) \downarrow g) \mid F(R) \downarrow g \text{ is non-empty} \}$.
416
 - 417 • A group cardinality (group count) report for one or more specified grouping
418 operators G_i , which may include, and may possibly be limited to, the default
419 (unnamed) group. In mathematical notation: $\{ (g, |F(R) \downarrow g|) \mid F(R) \downarrow g \text{ is non-} \}$
420 empty }.

421 **6 Read Cycle Timing**

422 The ALE API is intentionally silent about the timing of read cycles. Clients may specify
423 the boundaries of event cycles, which accumulate data from one or more underlying read
424 cycles, but the API does not provide a client with explicit control over the frequency at
425 which read cycles are completed. There are several reasons for this:

- A client or clients may make simultaneous requests for event cycle reports that may have differing event cycle boundaries and different report specifications. In this case, clients must necessarily share a common view of when and how frequently read cycles take place. Specifying the read cycle frequency outside of any event cycle request insures that clients cannot make contradictory demands on read cycles.
- In cases where there are many readers in physical proximity (perhaps communicating to different ALE implementations), the read cycle frequency must be carefully tuned and coordinated to avoid reader interference. This coordination generally requires physical-level information that generally would be (and should be) unknown to a client operating at the ALE level.
- The ALE API is designed to provide access to data from a wide variety of “Reader” sources, which may have very divergent operating principles. If the ALE API were to provide explicit control over read cycle timing, it would necessarily make assumptions about the source of read cycle data that would limit its applicability. For example, if the ALE API were to provide a parameter to clients to set the frequency of read cycles, it would assume that every Reader provides data on a fixed, regular schedule.

In light of these considerations, there is no standard way provided by ALE for clients to control read cycle timing. Implementations of ALE may provide different means for this, *e.g.*, configuration files, administrative interfaces, and so forth.

Regardless of how a given ALE implementation provides for the configuration of read cycle timing, the ALE implementation always has the freedom to suspend Reader activity during periods when no event cycles requiring data from a given Reader are active.

7 Logical Reader Names

In specifying an event cycle, an ALE client names one or more Readers of interest. This is usually necessary, as an ALE implementation may manage many readers that are used for unrelated purposes. For example, in a large warehouse, there may be ten loading dock doors each having three RFID readers; in such a case, a typical ALE request may be directed at the three readers for a particular door, but it is unlikely that an application tracking the flow of goods into trucks would want the reads from all 30 readers to be combined into a single event cycle.

This raises the question of how ALE clients specify which reader devices are to be used for a given event cycle. One possibility is to use identities associated with the reader devices themselves, *e.g.*, a unique name, serial number, EPC, IP address, *etc.* This is undesirable for several reasons:

- The exact identities of reader devices deployed in the field are likely to be unknown at the time an application is authored and configured.
- If a reader device is replaced, this unique reader device identity will change, forcing the application configuration to be changed.

- If the number of reader devices must change – *e.g.*, because it is discovered that four reader devices are required instead of three to obtain adequate coverage of a particular loading dock door – then the application must be changed.

To avoid these problems, ALE introduces the notion of a “logical reader.” Logical readers are abstract names that a client uses to refer to one or more Readers that have a single logical purpose; *e.g.*, `DockDoor42`. Within the implementation of ALE, an association is maintained between logical names such as `DockDoor42` and the physical reader devices assigned to fulfill that purpose. Any ALE event cycle specification that refers to `DockDoor42` is understood by the ALE implementation to refer to the physical reader (or readers) associated with that name.

Logical names may also be used to refer to sources of raw EPC events that are synthesized from various sources. For example, one vendor may have a technology for discriminating the physical location of tags by triangulating the results from several reader devices. This could be exposed in ALE by assigning a synthetic logical reader name for each discernable location.

Different ALE implementations may provide different ways of mapping logical names to physical reader devices, synthetic readers, and other sources of EPC events. This is a key extensibility point. At a minimum, however, all ALE implementations SHOULD provide a straightforward way to map a logical name to a list of read event sources, and where physical readers allow for independent control over multiple antennas and multiple tag protocols, each combination of (reader, antenna, protocol) should be treated as a separate read event source for this purpose. To illustrate, an ALE implementation may maintain a table like this:

Logical Reader Name	Physical Reader Devices		
	Reader Name	Antenna	Protocol
<code>DockDoor42</code>	<code>Acme42926</code>	0	UHF
	<code>Acme42926</code>	1	UHF
	<code>Acme43629</code>	0	UHF
<code>DockDoor43</code>	<code>Acme44926</code>	0	UHF
	<code>Acme44926</code>	1	UHF
	<code>Acme49256</code>	0	UHF

(It must be emphasized that the table above is meant to be illustrative of the kind of configuration data an ALE implementation might maintain, *not* a normative specification of what configuration data an ALE implementation must maintain.)

More elaborate implementations of ALE, such as those that provide synthesized logical readers such as the triangulation example above, will require more elaborate configuration data. Tables of this kind may be established through static configuration,

or through more dynamic discovery mechanisms. The method for establishing and maintaining configuration of this kind is outside the scope of this specification.

To summarize, the definition of ALE relies upon several related concepts:

- A *logical reader* is a name that an ALE client uses to refer to one or more, raw EPC data event sources (“Readers”). In terms of the formal model of Section 3, an event cycle aggregates read cycle data from all of the Readers that are associated with the set of logical readers the ALE client specifies in its request.
- A *Reader* is a raw EPC data event source. A Reader provides EPC data to an ALE implementation in a series of read cycles, each containing a list of EPCs. A Reader may map into physical devices in a variety of ways, including:
 - A Reader may map directly to a single physical device; *e.g.*, a one-antenna RFID reader, a bar code scanner, or a multi-antenna RFID reader where data from all antennas is always combined.
 - Several Readers may map to the same physical device; *e.g.*, a multi-antenna RFID reader where each antenna is treated as an independent source (in which case there would be a separate Reader for each antenna).
 - A Reader may map to more than one physical device; *e.g.*, several RFID devices are used to triangulate location information to create synthesized read cycles for virtual “Readers” associated with different spatial zones.

8 ALE API

This section defines normatively the programmatic interface to ALE. The external interface is defined by the ALE class (Section 8.1). This interface makes use of a number of complex data types that are documented in the sections following Section 8.1.

Implementations may expose the ALE interface via a wire protocol, or via a direct API in which clients call directly into code that implements ALE. This section of the document does not define the concrete wire protocol or programming language-specific API, but instead defines only the abstract syntax. Section 11 of the document specifies the required binding of the API to a WS-i compliant SOAP protocol. Section 10 specifies the standard way in which the two major data types in this API, the Event Cycle Specification and the Event Cycle Report, are rendered in XML. Implementations may provide additional bindings of the API, including bindings to particular programming languages, and of the data types.

The general interaction model is that there are one or more clients that make method calls to the ALE interface defined in Section 8.1. Each method call is a request, which causes the ALE implementation to take some action and return results. Thus, methods of the ALE interface are synchronous.

The ALE interface also provides a way for clients to subscribe to events that are delivered asynchronously. This is done through methods that take a `notificationURI` as an argument. Such methods return immediately, but subsequently the ALE implementation may asynchronously deliver information to the consumer denoted by the

535 notificationURI. Different ALE implementations MAY provide a variety of
536 available notification means (*e.g.*, JMS, MQ-Series, TIBCO, e-mail, SOAP, *etc.*); this is
537 intended to be a point of extensibility. Section 9 specifies notification means that are
538 standardized, and specifies the conformance requirement (MAY, SHOULD, SHALL) for
539 each.

540 In the sections below, the API is described using UML class diagram notation, like so:

```
541 dataMember1 : Type1  
542 dataMember2 : Type2  
543 ---  
544 method1 (ArgName:ArgType, ArgName:ArgType, ...) : ReturnType  
545 method2 (ArgName:ArgType, ArgName:ArgType, ...) : ReturnType
```

546 Within the UML descriptions, the notation <<extension point>> identifies a place
547 where implementations SHALL provide for extensibility through the addition of new
548 data members and/or methods. Extensibility mechanisms SHALL provide for both
549 proprietary extensions by vendors of ALE-compliant products, and for extensions defined
550 by EPCglobal through future versions of this specification or through new specifications.

551 In the case of the standard XML bindings for ECSpec and ECReports, the extension
552 points are implemented within the XML schema following the methodology described in
553 Section 10.1. In the case of the standard SOAP binding for the ALE interface, the
554 extension point is implemented simply by adding new operations to the WSDL.

8.1 ALE – Main API Class

```
---
define(specName:string, spec:ECSpec) : void
undefine(specName:string) : void
getECSpec(specName:string) : ECSpec
getECSpecNames() : List // returns a list of specNames as
strings
subscribe(specName:string, notificationURI:string) : void
unsubscribe(specName:string, notificationURI:string) : void
poll(specName:string) : ECReports
immediate(spec:ECSpec) : ECReports
getSubscribers(specName:String) : List // of notification
URIs
getStandardVersion() : string
getVendorVersion() : string
<<extension point>>
```

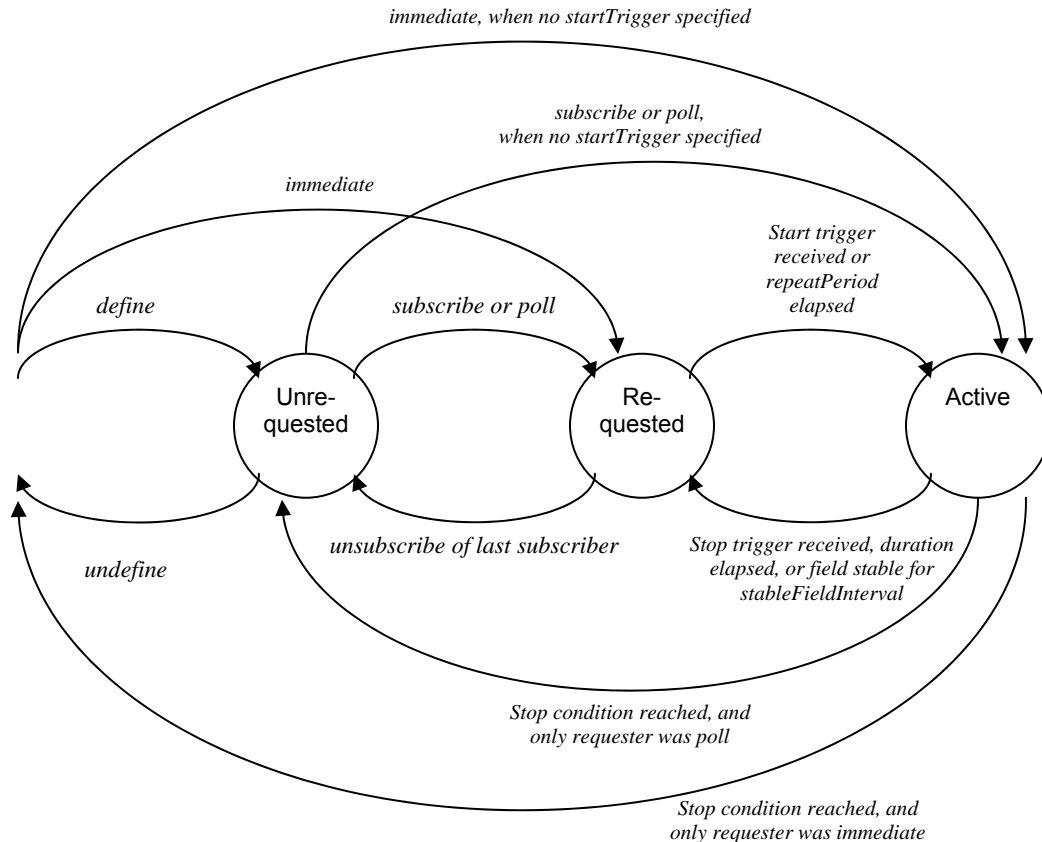
An ECSpec is a complex type that defines how an event cycle is to be calculated. There are two ways to cause event cycles to occur. A standing ECSpec may be posted using the `define` method. Subsequently, one or more clients may subscribe to that ECSpec using the `subscribe` method. The ECSpec will generate event cycles as long as there is at least one subscriber. A `poll` call is like subscribing then unsubscribing immediately after one event cycle is generated (except that the results are returned from `poll` instead of being sent to a `notificationURI`). The second way is that an ECSpec can be posted for immediate execution using the `immediate` method. This is equivalent to defining an ECSpec, performing a single `poll` operation, and then undefining it.

The execution of ECSpecs is defined formally as follows. An ECSpec is said to be *requested* if any of the following is true:

- It has previously been defined using `define`, it has not yet been undefined, and there has been at least one `subscribe` call for which there has not yet been a corresponding `unsubscribe` call.
- It has previously been defined using `define`, it has not yet been undefined, a `poll` call has been made, and the first event cycle since the `poll` was received has not yet been completed.
- It was defined using the `immediate` method, and the first event cycle has not yet been completed.

Once requested, an ECSpec is said to be *active* if reads are currently being accumulated into an event cycle based on the ECSpec. Standing ECSpecs that are requested using `subscribe` may transition between active and inactive multiple times. ECSpecs that are requested using `poll` or created using `immediate` will transition between active and inactive just once (though in the case of `poll`, the ECSpec remains defined afterward so that it could be subsequently polled again or subscribed to).

This description is summarized in the state diagram below.



The primary data types associated with the ALE API are the ECSpec, which specifies how an event cycle is to be calculated, and the ECRports, which contains one or more reports generated from one activation of an ECSpec. ECRports instances are both returned from the `poll` and `immediate` methods, and also sent to `notificationURIs` when ECSpecs are subscribed to using the `subscribe` method. The next two sections, Section 8.2 and Section 8.3, specify the ECSpec and ECRports data types in full detail.

The two methods `getStandardVersion` and `getVendorVersion` may be used by ALE clients to ascertain with what version of the ALE specification an implementation complies. The method `getStandardVersion` returns a string that identifies what version of the specification this implementation complies with. The possible values for this string are defined by EPCglobal. An implementation SHALL

611 return a string corresponding to a version of this specification to which the
612 implementation fully complies, and SHOULD return the string corresponding to the latest
613 version to which it complies. To indicate compliance with this Version 1.0 of the ALE
614 specification, the implementation SHALL return the string 1.0. The method
615 `getVendorVersion` returns a string that identifies what vendor extensions this
616 implementation provides. The possible values of this string and their meanings are
617 vendor-defined, except that the empty string SHALL indicate that the implementation
618 implements only standard functionality with no vendor extensions. When an
619 implementation chooses to return a non-empty string, the value returned SHALL be a
620 URI where the vendor is the owning authority. For example, this may be an HTTP URL
621 whose authority portion is a domain name owned by the vendor, a URN having a URN
622 namespace identifier issued to the vendor by IANA, an OID URN whose initial path is a
623 Private Enterprise Number assigned to the vendor, etc.

624 **8.1.1 Error Conditions**

625 Methods of the ALE API signal error conditions to the client by means of exceptions.
626 The following exceptions are defined. All the exception types in the following table are
627 extensions of a common `ALEException` base type, which contains one string element
628 giving the reason for the exception.

Exception Name	Meaning
<code>SecurityException</code>	The operation was not permitted due to an access control violation or other security concern. The specific circumstances that may cause this exception are implementation-specific, and outside the scope of this specification.
<code>DuplicateNameException</code>	The specified ECSpec name already exists.
<code>ECSpecValidationException</code>	The specified ECSpec is invalid; <i>e.g.</i> , it specifies both a start trigger and a repeat period. The complete list of rules for generating this exception are specified in Section 8.2.11.
<code>InvalidURIException</code>	The URI specified for a subscriber cannot be parsed, does not name a scheme recognized by the implementation, or violates rules imposed by a particular scheme.
<code>NoSuchNameException</code>	The specified ECSpec name does not exist.
<code>NoSuchSubscriberException</code>	The specified subscriber does not exist.

Exception Name	Meaning
DuplicateSubscriptionException	The specified ECSpec name and subscriber URI is identical to a previous subscription that was created and not yet unsubscribed.
ImplementationException	A generic exception thrown by the implementation for reasons that are implementation-specific. This exception contains one additional element: a severity member whose values are either ERROR or SEVERE. ERROR indicates that the ALE implementation is left in the same state it had before the operation was attempted. SEVERE indicates that the ALE implementation is left in an indeterminate state.

629

630 The exceptions that may be thrown by each ALE method are indicated in the table below:

ALE Method	Exceptions
define	DuplicateNameException ECSpecValidationException SecurityException ImplementationException
undefine	NoSuchNameException SecurityException ImplementationException
getECSpec	NoSuchNameException SecurityException ImplementationException
getECSpecNames	SecurityException ImplementationException
subscribe	NoSuchNameException InvalidURIException DuplicateSubscriptionException SecurityException ImplementationException
unsubscribe	NoSuchNameException NoSuchSubscriberException InvalidURIException SecurityException ImplementationException

ALE Method	Exceptions
poll	NoSuchNameException SecurityException ImplementationException
immediate	ECSpecValidationException SecurityException ImplementationException
getSubscribers	NoSuchNameException SecurityException ImplementationException

631

632 8.2 ECSpec

633 An ECSpec describes an event cycle and one or more reports that are to be generated
634 from it. It contains a list of logical Readers whose read cycles are to be included in the
635 event cycle, a specification of how the boundaries of event cycles are to be determined,
636 and a list of specifications each of which describes a report to be generated from this
637 event cycle.

```

638 readers : List    // List of logical reader names
639 boundaries : ECBoundarySpec
640 reportSpecs : List    // List of one or more ECRReportSpec
641                // instances
642 includeSpecInReports : boolean
643 <<extension point>>
644 ---

```

645 If the readers parameter is null, omitted, is an empty list, or contains any logical
646 reader names that are not known to the implementation, then the define and
647 immediate methods SHALL raise an ECSpecValidationException.

648 If the boundaries parameter is null or omitted, then the define and immediate
649 methods SHALL raise an ECSpecValidationException.

650 If the reportSpecs parameter is null or omitted or contains an empty list, or if the list
651 contains two ECRReportSpec instances with the same reportName, then the
652 define and immediate methods SHALL raise an
653 ECSpecValidationException.

654 If an ECSpec has includeSpecInReports set to true, then the ALE
655 implementation SHALL include the complete ECSpec as part of every ECRreports
656 instance generated by this ECSpec.

8.2.1 ECBoundarySpec

An ECBoundarySpec specifies how the beginning and end of event cycles are to be determined.

```
startTrigger : ECTrigger
repeatPeriod : ECTime
stopTrigger  : ECTrigger
duration     : ECTime
stableSetInterval : ECTime
<<extension point>>
---
```

The ECTime values duration, repeatPeriod, and stableSetInterval must be non-negative; otherwise, the define and immediate methods SHALL raise an ECTimeValidationException. Zero means “unspecified.”

The startTrigger and stopTrigger parameters are optional. For each of these two parameters, if the parameter is null, omitted, or is an empty string it is considered “unspecified.”

The startTrigger and repeatPeriod parameters are mutually exclusive. If startTrigger and repeatPeriod are both specified, then the define and immediate methods SHALL raise an ECTimeValidationException.

The conditions under which an event cycle is started depends on the settings for startTrigger and repeatPeriod:

- If startTrigger is specified and repeatPeriod is not specified, an event cycle is started when:
 - The ECTime is in the *requested* state and the specified start trigger is received.
- If startTrigger is not specified and repeatPeriod is specified, an event cycle is started when:
 - The ECTime transitions from the *unrequested* state to the *requested* state; or
 - The repeatPeriod has elapsed from the start of the last event cycle, and in that interval the ECTime has never transitioned to the *unrequested* state.
- If neither startTrigger nor repeatPeriod are specified, an event cycle is started when:
 - The ECTime transitions from the *unrequested* state to the *requested* state; or
 - Immediately after the previous event cycle, if the ECTime is in the *requested* state.

An event cycle, once started, extends until one of the following is true:

692 • The `duration`, when specified, expires.

693 • When the `stableSetInterval` is specified, no *new* EPCs are reported by any
694 Reader for the specified interval (*i.e.*, the set of EPCs being accumulated by the event
695 cycle is stable for the specified interval). In this context, “new” is to be interpreted
696 collectively among Readers contributing to this event cycle. For example, suppose a
697 given event cycle is accumulating data from Readers A and B. If Reader A completes
698 a read cycle containing EPC X, then subsequently Reader B completes a different
699 read cycle containing the same EPC X, then the occurrence of EPC X in B’s read
700 cycle is not considered “new” for the purposes of evaluating the
701 `stableSetInterval`. Note that in the context of the `stableSetInterval`,
702 the term “stable” only implies that no *new* EPCs are detected; it does not imply that
703 previously detected EPCs must continue to be detected. That is, only *additions*, and
704 not *deletions*, are considered in determining that the EPC set is “stable.”

705 • The `stopTrigger`, when specified, is received.

706 • The ECSpec transitions to the *unrequested* state.

707 Note that the first of these conditions to become true terminates the event cycle. For
708 example, if both `duration` and `stableSetInterval` are specified, then the event
709 cycle terminates when the `duration` expires, even if the reader field has not been stable
710 for the `stableSetInterval`. But if the set of EPCs is stable for
711 `stableSetInterval`, the event cycle terminates even if the total time is shorter than
712 the specified `duration`.

713 Note that if the `repeatPeriod` expires while an event cycle is in progress, it does not
714 terminate the event cycle. The event cycle terminates only when one of the four
715 conditions specified above becomes true. If, by that time, the ECSpec has not
716 transitioned to the *unrequested* state, then a new event cycle will start immediately,
717 following the second rule for `repeatPeriod` (because the `repeatPeriod` has
718 expired, the start condition is immediately fulfilled).

719 If no event cycle termination condition is specified in the `ECBoundarySpec` – that is,
720 `stopTrigger`, `duration`, and `stableSetInterval` are all unspecified, and
721 there is no vendor extension termination condition specified – then the `define` and
722 `immediate` methods SHALL raise an `ECSpecValidationException`.

723 In all the descriptions above, note that an ECSpec presented via the `immediate` method
724 means that the ECSpec transitions from *unrequested* to *requested* immediately upon
725 calling `immediate`, and transitions from *requested* to *unrequested* immediately after
726 completion of the event cycle.

727 The `ECTrigger` values `startTrigger` and `stopTrigger`, if specified, must
728 conform to URI syntax as defined by [RFC2396], and must be supported by the ALE
729 implementation; otherwise, the `define` and `immediate` methods SHALL raise an
730 `ECSpecValidationException`.

8.2.2 ECTime

ECTime denotes a span of time measured in physical time units.

```
duration : long
unit : ECTimeUnit
---
```

8.2.3 ECTimeUnit

ECTimeUnit is an enumerated type denoting different units of physical time that may be used in an ECBoundarySpec.

```
<<Enumerated Type>>
MS // Milliseconds
```

8.2.4 ECTrigger

ECTrigger denotes a URI that is used to specify a start or stop trigger for an event cycle (see Section 8.2.1 for explanation of start and stop triggers). The interpretation of this URI is determined by the ALE implementation; the kinds and means of triggers supported is intended to be a point of extensibility.

8.2.5 ECReportSpec

An ECReportSpec specifies one report to be returned from executing an event cycle. An ECSpec contains a list of one or more ECReportSpec instances.

```
reportName : string
reportSet : ECReportSetSpec
filter : ECFilterSpec
group : ECGroupSpec
output : ECReportOutputSpec
reportIfEmpty : boolean
reportOnlyOnChange : boolean
<<extension point>>
---
```

The ECReportSetSpec specifies what set of EPCs is considered for reporting: all currently read, additions from the previous event cycle, or deletions from the previous event cycle.

The filter parameter (of type ECFilterSpec) specifies how the raw EPCs are filtered before inclusion in the report. If any of the specified filters does not conform to

763 the EPC URI pattern syntax specified in [TDS1.1], then the `define` and `immediate`
764 methods SHALL raise an `ECSpecValidationException`.

765 The `group` parameter (of type `ECGroupSpec`) specifies how the filtered EPCs are
766 grouped together for reporting. If any of the grouping patterns does not conform to the
767 syntax for grouping patterns specified in Section 8.2.9, or if any two grouping patterns
768 are determined to be non-disjoint as defined in Section 8.2.9, then the `define` and
769 `immediate` methods SHALL raise an `ECSpecValidationException`.

770 The `output` parameter (of type `ECReportOutputSpec`) specifies whether to return
771 the EPC groups themselves or a count of each group, or both. These parameter types are
772 discussed at length in Sections 4 and 5.

773 If an `ECReportSpec` has `reportIfEmpty` set to `false`, then the corresponding
774 `ECReport` instance SHALL be omitted from the `ECReports` for this event cycle if the
775 final, filtered set of EPCs is empty (i.e., if the final EPC list would be empty, or if the
776 final count would be zero).

777 If an `ECReportSpec` has `reportOnlyOnChange` set to `true`, then the corresponding
778 `ECReport` instance SHALL be omitted from the `ECReports` for this event cycle if the
779 filtered set of EPCs is identical to the previously filtered set of EPCs. This comparison
780 takes place before the filtered set has been modified based on `reportSet` or `output`
781 parameters. The comparison also disregards whether the previous `ECReports` was
782 actually sent due to the effect of this boolean, or the `reportIfEmpty` boolean.

783 When the processing of `reportIfEmpty` and `reportOnlyOnChange` results in *all*
784 `ECReport` instances being omitted from an `ECReports` for an event cycle, then the
785 notification of subscribers SHALL be suppressed altogether. That is, a notification
786 consisting of an `ECReports` having zero contained `ECReport` instances SHALL NOT
787 be sent to a subscriber. (Because an `ECSpec` must contain at least one
788 `ECReportSpec`, this can only arise as a result of `reportIfEmpty` or
789 `reportOnlyOnChange` processing.) This rule only applies to subscribers (event cycle
790 requestors that were registered by use of the `subscribe` method); an `ECReports`
791 instance SHALL always be returned to the caller of `immediate` or `poll` at the end of
792 an event cycle, even if that `ECReports` instance contains zero `ECReport` instances.

793 The `reportName` parameter is an arbitrary string that is copied to the `ECReport`
794 instance created when this event cycle completes. The purpose of the `reportName`
795 parameter is so that clients can distinguish which of the `ECReport` instances that it
796 receives corresponds to which `ECReportSpec` instance contained in the original
797 `ECSpec`. This is especially useful in cases where fewer reports are delivered than there
798 were `ECReportSpec` instances in the `ECSpec`, because `reportIfEmpty=false`
799 or `reportOnlyOnChange=true` settings suppressed the generation of some reports.

8.2.6 ECRptSetSpec

ECRptSetSpec is an enumerated type denoting what set of EPCs is to be considered for filtering and output: all EPCs read in the current event cycle, additions from the previous event cycle, or deletions from the previous event cycle.

```
<<Enumerated Type>>
```

```
CURRENT
```

```
ADDITIONS
```

```
DELETIONS
```

8.2.7 ECFilterSpec

An ECFilterSpec specifies what EPCs are to be included in the final report.

```
includePatterns : List      // List of EPC patterns
```

```
excludePatterns : List      // List of EPC patterns
```

```
<<extension point>>
```

```
---
```

The ECFilterSpec implements a flexible filtering scheme based on two pattern lists. Each list contains zero or more EPC patterns. Each EPC pattern denotes a single EPC, a range of EPCs, or some other set of EPCs. (Patterns are described in detail below in Section 8.2.8.) An EPC is included in the final report if (a) the EPC does *not* match any pattern in the excludePatterns list, *and* (b) the EPC *does* match at least one pattern in the includePatterns list. The (b) test is omitted if the includePatterns list is empty.

This can be expressed using the notation of Section 4 as follows, where R is the set of EPCs to be reported from a given event cycle, prior to filtering:

$$F(R) = \{ epc \mid epc \in R \\ \& (epc \in I_1 \mid \dots \mid epc \in I_n) \\ \& epc \notin E_1 \& \dots \& epc \notin E_n \}$$

where I_i denotes the set of EPCs matched by the i th pattern in the includePatterns list, and E_i denotes the set of EPCs matched by the i th pattern in the excludePatterns list.

8.2.8 EPC Patterns (non-normative)

EPC Patterns are used to specify filters within an ECFilterSpec. The normative specification of EPC Patterns may be found in the EPCglobal Tag Data Specification Version 1.1 [TDS1.1]. The remainder of this section provides a non-normative summary of some of the features of that specification, to aid the reader who has not read the EPCglobal Tag Data Specification in understanding the filtering aspects of the ALE API.

835 An EPC pattern is a URI-formatted string that denotes a single EPC or set of EPCs. The
836 general format is:

837 `urn:epc:pat:TagFormat:Filter.Company.Item.Serial`

838 where *TagFormat* denotes one of the tag formats defined by the Tag Data
839 Specification, and the four fields *Filter*, *Company*, *Item*, and *SerialNumber*
840 correspond to data fields of the EPC. The meaning and number of these fields, as well as
841 their formal names, varies according to what *TagFormat* is named. In an EPC pattern,
842 each of the data fields may be (a) a decimal integer, meaning that a matching EPC must
843 have that specific value in the corresponding field; (b) an asterisk (*), meaning that a
844 matching EPC may have any value in that field; or (c) a range denoted like [*lo-hi*],
845 meaning that a matching EPC must have a value between the decimal integers *lo* and
846 *hi*, inclusive. Depending on the tag format, there may be other restrictions; see the
847 EPCglobal Tag Data Specification for full details.

848 Here are some examples. In these examples, assume that all tags are of the GID-96
849 format (which lacks the *Filter* data field), and that 20 is the Domain Manager
850 (*Company* field) for XYZ Corporation, and 300 is the Object Class (*Item* field) for its
851 UltraWidget product.

<code>urn:epc:pat:gid-96:20.300.4000</code>	Matches the EPC for UltraWidget serial number 4000.
<code>urn:epc:pat:gid-96:20.300.*</code>	Matches any UltraWidget's EPC, regardless of serial number.
<code>urn:epc:pat:gid-96:20.*.[5000-9999]</code>	Matches any XYZ Corporation product whose serial number is between 5000 and 9999, inclusive.
<code>urn:epc:pat:gid-96:*.*.*</code>	Matches any GID-96 tag

852

853 **8.2.9 ECGroupSpec**

854 ECGroupSpec defines how filtered EPCs are grouped together for reporting.

855 `patternList : List // of pattern URIs`
856 `---`

857 Each element of the pattern list is an EPC Pattern URI as defined by the EPCglobal Tag
858 Data Specification Version 1.1 [TDS1.1] (see Section 8.2.8 for an informal description of
859 this syntax), extended by allowing the character X in each position where a * character is
860 allowed. All restrictions on the use of the * character as defined in the Tag Data
861 Specification apply equally to the use of the X character. For example, the following are
862 legal URIs for use in the pattern list:

863 `urn:epc:pat:sgtin-64:3.*.*.*`
864 `urn:epc:pat:sgtin-64:3.*.X.*`

865 urn:epc:pat:sgtin-64:3.X.*.*

866 urn:epc:pat:sgtin-64:3.X.X.*

867 But the following are not:

868 urn:epc:pat:sgtin-64:3.*.12345.*

869 urn:epc:pat:sgtin-64:3.X.12345.*

870 Pattern URIs used in an ECGroupSpec are interpreted as follows:

Pattern URI Field	Meaning
X	Create a different group for each distinct value of this field.
*	All values of this field belong to the same group.
<i>Number</i>	Only EPCs having <i>Number</i> in this field will belong to this group.
[<i>Lo-Hi</i>]	Only EPCs whose value for this field falls within the specified range will belong to this group.

871

872 Here are examples of pattern URIs used as group operators:

Pattern URI	Meaning
urn:epc:pat:sgtin-64:X.*.*.*	groups by filter value (<i>e.g.</i> , case/pallet)
urn:epc:pat:sgtin-64:*.X.*.*	groups by company prefix
urn:epc:pat:sgtin-64:*.X.X.*	groups by company prefix and item reference (<i>i.e.</i> , groups by specific product)
urn:epc:pat:sgtin-64:X.X.X.*	groups by company prefix, item reference, and filter
urn:epc:pat:sgtin-64:3.X.*.[0-100]	create a different group for each company prefix, including in each such group only EPCs having a filter value of 3 and serial numbers in the range 0 through 100, inclusive

873

874 In the corresponding ECGReport, each group is named by another EPC Pattern URI that
875 is identical to the group operator URI, except that the group name URI has an actual
876 value in every position where the group operator URI had an X character.

877 For example, if these are the filtered EPCs read for the current event cycle:

878 urn:epc:tag:sgtin-64:3.0036000.123456.400

879 urn:epc:tag:sgtin-64:3.0036000.123456.500

880 urn:epc:tag:sgtin-64:3.0029000.111111.100

881 urn:epc:tag:sscc-64:3.0012345.31415926

882 Then a pattern list consisting of just one element, like this:

883 urn:epc:pat:sgtin-64:*.X.*.*

884 would generate the following groups in the report:

Group Name	EPCs in Group
urn:epc:pat:sgtin-64:*.0036000.*.*	urn:epc:tag:sgtin-64:3.0036000.123456.400 urn:epc:tag:sgtin-64:3.0036000.123456.500
urn:epc:pat:sgtin-64:*.0029000.*.*	urn:epc:tag:sgtin-64:3.0029000.111111.100
[default group]	urn:epc:tag:sscc-64:3.0012345.31415926

885

886 Every filtered EPC that is part of the event cycle is part of exactly one group. If an EPC
887 does not match any of the EPC Pattern URIs in the pattern list, it is included in a special
888 “default group.” The name of the default group is null. In the above example, the SSCC
889 EPC did not match any pattern in the pattern list, and so was included in the default
890 group.

891 As a special case of the above rule, if the pattern list is empty (or if the group parameter
892 of the ECRptSpec is null or omitted), then all EPCs are part of the default group.

893 In order to insure that each EPC is part of only one group, there is an additional
894 restriction that all patterns in the pattern list must be pairwise disjoint. Disjointedness of
895 two patterns is defined as follows. Let Pat_i and Pat_j be two pattern URIs, written as a
896 series of fields as follows:

897 Pat_i = urn:epc:pat:type_i:field_{i_1}.field_{i_2}.field_{i_3}...

898 Pat_j = urn:epc:pat:type_j:field_{j_1}.field_{j_2}.field_{j_3}...

899 Then Pat_i and Pat_j are disjoint if:

- 900 • type_i is not equal to type_j
- 901 • type_i = type_j but there is at least one field *k* for which field_{i_k} and
- 902 field_{j_k} are disjoint, as defined by the table below:

	X	*	Number	[Lo-Hi]
X	Not disjoint	Not disjoint	Not disjoint	Not disjoint
*	Not disjoint	Not disjoint	Not disjoint	Not disjoint
Number	Not disjoint	Not disjoint	Disjoint if the numbers are different	Disjoint if the number is not included in the range
[Lo-Hi]	Not disjoint	Not disjoint	Disjoint if the number is not	Disjoint if the ranges do not

			included in the range	overlap
--	--	--	--------------------------	---------

903

904 The relationship of the ECGroupSpec to the group operator introduced in Section 5 is
905 defined as follows. Formally, a group operator G is specified by a list of pattern URIs:

906 $G = (\text{Pat_1}, \text{Pat_2}, \dots, \text{Pat_N})$

907 Let each pattern be written as a series of fields:

908 $\text{Pat_i} = \text{urn:epc:pat:type_i:field_i_1.field_i_2.field_i_3}\dots$

909 where each field_i_j is either X, *, *Number*, or [*Lo-Hi*].

910 Then the definition of G(epc) is as follows. Let epc be written like this:

911 $\text{urn:epc:tag:type_epc:field_epc_1.field_epc_2.field_epc_3}\dots$

912 The epc is said to *match* Pat_i if

- 913 • $\text{type_epc} = \text{type_i}$; and
- 914 • For each field k , one of the following is true:
 - 915 • $\text{field_i_k} = X$
 - 916 • $\text{field_i_k} = *$
 - 917 • field_i_k is a number, equal to field_epc_k
 - 918 • field_i_k is a range [*Lo-Hi*], and $Lo \leq \text{field_epc_k} \leq Hi$

919 Because of the disjointedness constraint specified above, the epc is guaranteed to match
920 at most one of the patterns in G.

921 G(epc) is then defined as follows:

- 922 • If epc matches Pat_i for some i, then
 - 923 $G(\text{epc}) = \text{urn:epc:pat:type_epc:field_g_1.field_g_2.field_g_3}\dots$
 - 924 where for each k , $\text{field_g_k} = *$, if $\text{field_i_k} = *$; or $\text{field_g_k} =$
925 field_epc_j , otherwise
- 926 • If epc does not match Pat_i for any i, then $G(\text{epc}) =$ the default group.

927 **8.2.10 ECRreportOutputSpec**

928 ECRreportOutputSpec specifies how the final set of EPCs is to be reported.

```

929 includeEPC : boolean
930 includeTag : boolean
931 includeRawHex : boolean
932 includeRawDecimal : boolean
933 includeCount : boolean
934 <<extension point>>
935 ---

```

936 If any of the four booleans `includeEPC`, `includeTag`, `includeRawHex`, or
 937 `includeRawDecimal` are true, the report SHALL include a list of the EPCs in the
 938 final set for each group. Each element of this list, when included, SHALL include the
 939 formats specified by these four Booleans. If `includeCount` is true, the report SHALL
 940 include a count of the EPCs in the final set for each group. Both may be true, in which
 941 case each group includes both a list and a count. If all five booleans `includeEPC`,
 942 `includeTag`, `includeRawHex`, `includeRawDecimal`, and `includeCount` are
 943 false, in the absence of any vendor extension to `ECReportOutputSpec`, then the
 944 define and immediate methods SHALL raise an
 945 `ECSpecValidationException`.

946 **8.2.11 Validation of ECSpecs**

947 The define and immediate methods of the ALE API (Section 8.1) SHALL raise an
 948 `ECSpecValidationException` if any of the following are true:

- 949 • Any logical reader name in the `readers` field of `ECSpec` is not known to the
 950 implementation.
- 951 • The `startTrigger` or `stopTrigger` field of `ECBoundarySpec`, when
 952 specified, does not conform to URI syntax as defined by [RFC2396], or is not
 953 supported by the ALE implementation.
- 954 • The `duration`, `stableSetInterval`, or `repeatPeriod` field of
 955 `ECBoundarySpec` is negative.
- 956 • The `startTrigger` field of `ECBoundarySpec` is non-empty *and* the
 957 `repeatPeriod` field of `ECBoundarySpec` is non-zero.
- 958 • No stopping condition is specified in `ECBoundarySpec`; *i.e.*, neither
 959 `stopTrigger` nor `duration` nor `stableSetInterval` nor any vendor
 960 extension stopping condition is specified.
- 961 • The list of `ECReportSpec` instances is empty.
- 962 • Two `ECReportSpec` instances have identical values for their `name` field.
- 963 • The `boundaries` parameter of `ECSpec` is null or omitted.

- 964 • Any filter within `ECFilterSpec` does not conform to the EPC URI pattern syntax
965 specified in [TDS1.1].
- 966 • Any grouping pattern within `ECGroupSpec` does not conform to the syntax for
967 grouping patterns specified in Section 8.2.9.
- 968 • Any two grouping patterns within `ECGroupSpec` are determined to be non-disjoint
969 as that term is defined in Section 8.2.9.
- 970 • Within any `ECReportSpec` of an `ECSpec`, the `ECReportOutputSpec` has no
971 output type specified; *i.e.*, none of `includeEPC`, `includeTag`,
972 `includeRawHex`, `includeRawDecimal`, `includeCount`, nor any vendor
973 extension output type is specified as true.

974 8.3 ECReports

975 `ECReports` is the output from an event cycle.

```
976 specName : string
977 date : dateTime
978 ALEID : string
979 totalMilliseconds : long
980 terminationCondition : ECTerminationCondition
981 spec : ECSpec
982 reports : List    // List of ECReport
983 <<extension point>>
984 ---
```

985 The “meat” of an `ECReports` instance is the list of `ECReport` instances, each
986 corresponding to an `ECReportSpec` instance in the event cycle’s `ECSpec`. In addition
987 to the reports themselves, `ECReports` contains a number of “header” fields that provide
988 useful information about the event cycle:

Field	Description
<code>specName</code>	The name of the <code>ECSpec</code> that controlled this event cycle. In the case of an <code>ECSpec</code> that was requested using the <code>immediate</code> method (Section 8.1), this name is one chosen by the ALE implementation.
<code>date</code>	A representation of the date and time when the event cycle ended. For bindings in which this field is represented textually, an ISO-8601 compliant representation SHOULD be used.
<code>ALEID</code>	An identifier for the deployed instance of the ALE implementation. The meaning of this identifier is

Field	Description
	outside the scope of this specification.
totalMilliseconds	The total time, in milliseconds, from the start of the event cycle to the end of the event cycle.
terminationCondition	Indicates what kind of event caused the event cycle to terminate: the receipt of an explicit stop trigger, the expiration of the event cycle duration, or the read field being stable for the prescribed amount of time. These correspond to the possible ways of specifying the end of an event cycle as defined in Section 8.2.1.
spec	A copy of the ECSpec that generated this EReports instance. Only included if the ECSpec has includeSpecInReports set to true.

989

990 **8.3.1 ECTerminationCondition**

991 ECTerminationCondition is an enumerated type that describes how an event cycle
992 was ended.

993	<<Enumerated Type>>
994	TRIGGER
995	DURATION
996	STABLE_SET
997	UNREQUEST

998 The first three values, TRIGGER, DURATION, and STABLE_SET, correspond to the
999 receipt of an explicit stop trigger, the expiration of the event cycle duration, or the set
1000 of EPCs being stable for the event cycle stableSetInterval, respectively. These
1001 are the possible stop conditions described in Section 8.2.1. The last value, UNREQUEST,
1002 corresponds to an event cycle being terminated because there were no longer any clients
1003 requesting it. By definition, this value cannot actually appear in an EReports
1004 instance sent to any client.

1005 **8.3.2 ECRReport**

1006 ECRReport represents a single report within an event cycle.

```

1007  reportName : string
1008  groups : List      // List of ECReportGroup instances
1009  <<extension point>>
1010  ---

```

1011 The reportName field is a copy of the reportName field from the corresponding
 1012 ECReportSpec within the ECSpec that controlled this event cycle. The groups
 1013 field is a list containing one element for each group in the report as controlled by the
 1014 group field of the corresponding ECReportSpec. When no grouping is specified, the
 1015 groups list just consists of the single default group.

1016 **8.3.3 ECReportGroup**

1017 ECReportGroup represents one group within an ECReport.

```

1018  groupName : string
1019  groupList : ECReportGroupList
1020  groupCount : ECReportGroupCount
1021  <<extension point>>
1022  ---

```

1023 The groupName SHALL be null for the default group. The groupList field SHALL
 1024 be null if the includeEPC, includeTag, includeRawHex, and
 1025 includeRawDecimal fields of the corresponding ECReportOutputSpec are all
 1026 false (unless ECReportOutputSpec has vendor extensions that cause groupList
 1027 to be included). The groupCount field SHALL be null if the includeCount field
 1028 of the corresponding ECReportOutputSpec is false (unless
 1029 ECReportOutputSpec has vendor extensions that cause groupCount to be
 1030 included).

1031 **8.3.4 ECReportGroupList**

1032 An ECReportGroupList SHALL be included in an ECReportGroup when any of
 1033 the four boolean fields includeEPC, includeTag, includeRawHex, and
 1034 includeRawDecimal of the corresponding ECReportOutputSpec are true.

```

1035  members : List //List of ECReportGroupListMember instances
1036  <<extension point>>
1037  ---

```

1038 The order in which EPCs are enumerated within the list is unspecified.

8.3.5 ECRptGroupListMember

Each member of the `ECRptGroupList` is an `ECRptGroupListMember` as defined below. The reason for having `ECRptGroupListMember` is to allow multiple EPC formats to be included, and to provide an extension point for adding per-EPC information to the list report.

```
epc : URI
tag : URI
rawHex : URI
rawDecimal : URI
<<extension point>
---
```

Each of these fields SHALL contain a URI as described below or be null, depending on the value of a boolean in the corresponding `ECRptOutputSpec`. Specifically, the `epc` field SHALL be non-null if and only if the `includeEPC` field of `ECRptOutputSpec` is true, the `tag` field SHALL be non-null according to `includeTag`, the `rawHex` field SHALL be non-null according to `includeRawHex`, and the `rawDecimal` field SHALL be non-null according to `includeDecimal`.

When non-null, the `epc` field SHALL contain an EPC represented as a pure identity URI according to the EPCglobal Tag Data Specification (`urn:epc:id:...`). This URI SHALL be determined using the first procedure given in Section 5 of [TDS1.1]. If that procedure fails in any step, the `epc` field SHALL instead contain a raw decimal URI determined using Step 20 of the second procedure given in Section 5 of [TDS1.1].

When non-null, the `tag` field SHALL contain an EPC represented as a tag URI according to the EPCglobal Tag Data Specification (`urn:epc:tag:...`). This URI SHALL be determined using the second procedure given in Section 5 of [TDS1.1].

When non-null, the `rawDecimal` field SHALL contain a raw tag value represented as a raw decimal URI according to the EPCglobal Tag Data Specification (`urn:epc:raw:...`). This URI SHALL be determined using Step 20 of the second procedure given in Section 5 of [TDS1.1].

When non-null, the `rawHex` field SHALL contain a raw tag value represented as a raw hexadecimal URI according to the following extension to the EPCglobal Tag Data Specification. The URI SHALL be determined by concatenating the following: the string `urn:epc:raw:`, the length of the tag value in bits, a dot (.) character, a lowercase `x` character, and the tag value considered as a single hexadecimal integer. The length value preceding the dot character SHALL have no leading zeros. The hexadecimal tag value following the dot SHALL have a number of characters equal to the length of the tag value in bits divided by four and rounded up to the nearest whole number, and SHALL only use uppercase letters for the hexadecimal digits A, B, C, D, E, and F.

Each distinct tag value included in the report SHALL have a distinct ECRptGroupListMember element in the ECRptGroupList, even if those ECRptGroupListMember elements would be identical due to the formats selected. In particular, it is possible for two different tags to have the same pure identity EPC representation; e.g., two SGTIN-64 tags that differ only in the filter bits. If both tags are read in the same event cycle, and ECRptOutputSpec specified includeEPC true and all other formats false, then the resulting ECRptGroupList SHALL have two ECRptGroupListMember elements, each having the same pure identity URI in the epc field. In other words, the result should be equivalent to performing all duplicate removal, additions/deletions processing, grouping, and filtering *before* converting the raw tag values into the selected representation(s).

Explanation (non-normative): The situation in which this rule applies is expected to be extremely rare. In theory, no two tags should be programmed with the same pure identity, even if they differ in filter bits or other fields not part of the pure identity. But because the situation is possible, it is necessary to specify a definite behavior in this specification. The behavior specified above is intended to be the most easily implemented.

8.3.6 ECRptGroupCount

An ECRptGroupCount is included in an ECRptGroup when the includeCount field of the corresponding ECRptOutputSpec is true.

```
count : int
<<extension point>>
---
```

The count field is the total number of distinct EPCs that are part of this group.

9 Standard Notification URIs

This section specifies the syntax and semantics of standard URIs that may be used in conjunction with the subscribe and unsubscribe methods of the main ALE interface (Section 8.1). Each subsection below specifies the conformance requirement (MAY, SHOULD, SHALL) for each standard URI.

All notification URIs, whether standardized as a part of this specification or not, must conform to the general syntax for URIs as defined in [RFC2396]. Each notification URI scheme may impose additional constraints upon syntax.

9.1 HTTP Notification URI

The HTTP notification URI provides for delivery of ECRpts in XML via the HTTP protocol using the POST operation. Implementations SHOULD provide support for this notification URI.

1115 The syntax for HTTP notification URIs as used by ALE is defined in [RFC2616],
1116 Section 3.2.2. Informally, an HTTP URI has one of the two following forms:
1117 <http://host:port/remainder-of-URL>
1118 `http://host/remainder-of-URL`
1119 where
1120 • *host* is the DNS name or IP address of the host where the receiver is listening for
1121 incoming HTTP connections.
1122 • *port* is the TCP port on which the receiver is listening for incoming HTTP
1123 connections. The port and the preceding colon character may be omitted, in which
1124 case the port defaults to 80.
1125 • *remainder-of-URL* is the URL to which an HTTP POST operation will be
1126 directed.
1127 The ALE implementation delivers event cycle reports by sending an HTTP POST request
1128 to receiver designated in the URI, where *remainder-of-URL* is included in the HTTP
1129 request-line (as defined in [RFC2616]), and where the payload is the `ECReports`
1130 instance encoded in XML according to the schema specified in Section 10.2.
1131 The interpretation by the ALE implementation of the response code returned by the
1132 receiver is outside the scope of this specification; however, all implementations SHALL
1133 interpret a response code 2xx (that is, any response code between 200 and 299, inclusive)
1134 as a normal response, not indicative of any error.

1135 **9.2 TCP Notification URI**

1136 The TCP notification URI provides for delivery of `ECReports` in XML via a raw TCP
1137 connection. Implementations SHOULD provide support for this notification URI.

1138 The syntax for TCP notification URIs as used by ALE is as follows:

1139 `tcp_URL = "tcp:" "/" host ":" port`

1140 where the syntax definition for `host` and `port` is specified in [RFC2396].

1141 Informally, a TCP URI has the following form:

1142 `tcp://host:port`

1143 The ALE implementation delivers an event cycle report by opening a new TCP
1144 connection to the specified host and port, writing to the connection the `ECReports`
1145 instance encoded in XML according to the schema specified in Section 10.2, and then
1146 closing the connection. No reply or acknowledgement is expected by the ALE
1147 implementation.

1148 **9.3 FILE Notification URI**

1149 The FILE notification URI provides for writing of `ECReports` in XML to a file.
1150 Implementations MAY provide support for this notification URI.

1151 The syntax for FILE notification URIs as used by ALE is defined in [RFC1738],
1152 Section 3.10. Informally, an FILE URI has one of the two following forms:
1153 `file://host/path`
1154 `file:///path`
1155 where

- 1156 • `host` is the DNS name or IP address of a remote host whose filesystem is accessible
1157 to the ALE implementation.
- 1158 • `path` is the pathname of a file within the remote filesystem, or the local filesystem if
1159 `host` is omitted.

1160 The ALE implementation delivers an event cycle report by appending to the specified file
1161 the ECRports instance encoded in XML according to the schema specified in
1162 Section 10.2. Note that if more than one event cycle completes, the file will contain a
1163 concatenation of XML documents, rather than a single XML document.

1164 Implementations of ALE may impose additional constraints on the use of the FILE URI.
1165 For example, some implementations of ALE may support only a local filesystem while
1166 others may support only a remote filesystem, some implementations of ALE may impose
1167 further restrictions on the syntax of the `path` component, and so forth. This
1168 specification also does not define the behavior when `path` names a directory; the
1169 behavior in that case is implementation dependent.

1170 *Rationale (non-normative): The intended use for the FILE notification URI is for*
1171 *debugging, and hence the specification is intentionally lax in order to give freedom to*
1172 *implementations to provide the most appropriate and useful facility given the unique*
1173 *circumstances of that implementation.*

1174 **10 XML Schema for Event Cycle Specs and Reports**

1175 This section defines the standard XML representation for ECSpec instances
1176 (Section 8.2) and ECRports instances (Section 8.3), using the W3C XML Schema
1177 language [XSD1, XSD2]. Samples are also shown.

1178 The schema below conforms to EPCglobal standard schema design rules. The schema
1179 below imports the EPCglobal standard base schema, as mandated by the design rules.

1180 **10.1 Extensibility Mechanism**

1181 The XML schema in this section implements the <<extension point>> given in
1182 the UML of Section 8 using a methodology described in [XMLVersioning]. This
1183 methodology provides for both vendor extension, and for extension by EPCglobal in
1184 future versions of this specification or in supplemental specifications. Extensions
1185 introduced through this mechanism will be *backward compatible*, in that documents
1186 conforming to older versions of the schema will also conform to newer versions of the
1187 standard schema and to schema containing vendor-specific extensions. Extensions will
1188 also be *forward compatible*, in that documents that contain vendor extensions or that

1189 conform to newer versions of the standard schema will also conform to older versions of
1190 the schema.

1191 When a document contains extensions (vendor-specific or standardized in newer versions
1192 of schema), it may conform to more than one schema. For example, a document
1193 containing vendor extensions to the EPCglobal Version 1.0 schema will conform both to
1194 the EPCglobal Version 1.0 schema and to a vendor-specific schema that includes the
1195 vendor extensions. In this example, when the document is parsed using the standard
1196 schema there will be no type-checking of the extension elements and attributes, but when
1197 the document is parsed using the vendor-specific schema the extensions will be type-
1198 checked. Similarly, a document containing new features introduced in a hypothetical
1199 EPCglobal Version 1.1 schema will conform both to the EPCglobal Version 1.0 schema
1200 and to the EPCglobal Version 1.1 schema, but type checking of the new features will
1201 only be available using the Version 1.1 schema.

1202 The design rules for this extensibility pattern are given in [XMLVersioning]. In
1203 summary, it amounts to the following rules:

- 1204 • For each type in which <<extension point>> occurs, include an
1205 `xsd:anyAttribute` declaration. This declaration provides for the addition of
1206 new attributes, either in subsequent versions of the standard schema or in vendor-
1207 specific schema.
- 1208 • For each type in which <<extension point>> occurs, include an optional
1209 (`minOccurs = 0`) element named `extension`. The type declared for the
1210 `extension` element will always be as follows:

```
1211 <xsd:sequence>  
1212   <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"  
1213     namespace="##local"/>  
1214 </xsd:sequence>  
1215 <xsd:anyAttribute processContents="lax"/>
```

1216 This declaration provides for forward-compatibility with new elements introduced
1217 into subsequent versions of the standard schema.

- 1218 • For each type in which <<extension point>> occurs, include at the end of the
1219 element list a declaration

```
1220 <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"  
1221   namespace="##other"/>
```

1222 This declaration provides for forward-compatibility with new elements introduced in
1223 vendor-specific schema.

1224 The rules for adding vendor-specific extensions to the schema are as follows:

- 1225 • Vendor-specific attributes may be added to any type in which <<extension
1226 `point>>` occurs. Vendor-specific attributes SHALL NOT be in the EPCglobal ALE
1227 namespace (`urn:epcglobal:ale:xsd:1`). Vendor-specific attributes SHALL
1228 be in a namespace whose namespace URI has the vendor as the owning authority. (In
1229 schema parlance, this means that all vendor-specific attributes must have
1230 qualified as their form.) For example, the namespace URI may be an HTTP
1231 URL whose authority portion is a domain name owned by the vendor, a URN having

1232 a URN namespace identifier issued to the vendor by IANA, an OID URN whose
1233 initial path is a Private Enterprise Number assigned to the vendor, etc. Declarations
1234 of vendor-specific attributes SHALL specify use="optional".

- 1235 • Vendor-specific elements may be added to any type in which <<extension
1236 point>> occurs. Vendor-specific elements SHALL NOT be in the EPCglobal ALE
1237 namespace (urn:epcglobal:ale:xsd:1). Vendor-specific attributes SHALL
1238 be in a namespace whose namespace URI has the vendor as the owning authority (as
1239 described above). (In schema parlance, this means that all vendor-specific elements
1240 must have qualified as their form.)

1241 To create a schema that contains vendor extensions, replace the <xsd:any ...
1242 namespace="##other"/> declaration with a content group reference to a group
1243 defined in the vendor namespace; e.g., <xsd:group
1244 ref="vendor:VendorExtension">. In the schema file defining elements for
1245 the vendor namespace, define a content group using a declaration of the following
1246 form:

```
1247 <xsd:group name="VendorExtension">  
1248   <xsd:sequence>  
1249     <!--  
1250       Definitions or references to vendor elements  
1251       go here. Each SHALL specify minOccurs="0".  
1252     -->  
1253     <xsd:any processContents="lax"  
1254           minOccurs="0" maxOccurs="unbounded"  
1255           namespace="##other"/>  
1256   </xsd:sequence>  
1257 </xsd:group>
```

1258 (In the foregoing illustrations, vendor and VendorExtension may be any
1259 strings the vendor chooses.)

1260 *Explanation (non-normative): Because vendor-specific elements must be optional,*
1261 *including references to their definitions directly into the ALE schema would violate the*
1262 *XML Schema Unique Particle Attribution constraint, because the <xsd:any ...>*
1263 *element in the ALE schema can also match vendor-specific elements. Moving the*
1264 *<xsd:any ...> into the vendor's schema avoids this problem, because ##other in*
1265 *that schema means "match an element that has a namespace other than the vendor's*
1266 *namespace." This does not conflict with standard elements, because the element form*
1267 *default for the standard ALE schema is unqualified, and hence the ##other in the*
1268 *vendor's schema does not match standard ALE elements, either.*

1269 The rules for adding attributes or elements to future versions of the EPCglobal standard
1270 schema are as follows:

- 1271 • Standard attributes may be added to any type in which <<extension point>>
1272 occurs. Standard attributes SHALL NOT be in any namespace, and SHALL NOT
1273 conflict with any existing standard attribute name.

- Standard elements may be added to any type in which <<extension point>> occurs. New elements are added using the following rules:
 - Find the innermost extension element type.
 - Replace the <xsd:any ... namespace="##local"/> declaration with (a) new elements (which SHALL NOT be in any namespace); followed by (b) a new extension element whose type is constructed as described before. In subsequent revisions of the standard schema, new standard elements will be added within this new extension element rather than within this one.

Explanation (non-normative): the reason that new standard attributes and elements are specified above not to be in any namespace is to be consistent with the ALE schema's attribute and element form default of unqualified.

10.2 Schema

The following is an XML Schema (XSD) defining both ECSpec and ECREports.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="urn:epcglobal:ale:xsd:1"
  xmlns:ale="urn:epcglobal:ale: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 (C) 2005, 2004 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>
        Application Level Events (ALE) version 1.0
      </epcglobal:specification>
    </xsd:documentation>
  </xsd:annotation>

  <xsd:import namespace="urn:epcglobal:xsd:1" schemaLocation="./EpcGlobal.xsd"/>

  <!-- ALE ELEMENTS -->

  <xsd:element name="ECSpec" type="ale:ECSpec"/>
  <xsd:element name="ECReports" type="ale:ECReports"/>

  <!-- ALE TYPES -->

  <!-- items listed alphabetically by name -->

  <!-- Some element types accommodate extensibility in the manner of
  "Versioning XML Vocabularies" by David Orchard (see
  http://www.xml.com/pub/a/2003/12/03/versioning.html).

  In this approach, an optional <extension> element is defined
  for each extensible element type, where an <extension> element
```

may contain future elements defined in the target namespace.

In addition to the optional <extension> element, extensible element types are declared with a final xsd:any wildcard to accommodate future elements defined by third parties (as denoted by the ##other namespace).

Finally, the xsd:anyAttribute facility is used to allow arbitrary attributes to be added to extensible element types. -->

```
<xsd:complexType name="ECBoundarySpec">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      A ECBoundarySpec specifies how the beginning and end of event cycles
      are to be determined. The startTrigger and repeatPeriod elements
      are mutually exclusive. One may, however, specify a ECBoundarySpec
      with neither event cycle start condition (i.e., startTrigger nor
      repeatPeriod) present. At least one event cycle stopping condition
      (stopTrigger, duration, and/or stableSetInterval) must be present.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="startTrigger" type="ale:ECTrigger" minOccurs="0"/>
    <xsd:element name="repeatPeriod" type="ale:ECTime" minOccurs="0"/>
    <xsd:element name="stopTrigger" type="ale:ECTrigger" minOccurs="0"/>
    <xsd:element name="duration" type="ale:ECTime" minOccurs="0"/>
    <xsd:element name="stableSetInterval" type="ale:ECTime" minOccurs="0"/>
    <xsd:element name="extension" type="ale:ECBoundarySpecExtension"
      minOccurs="0"/>
    <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
      namespace="##other"/>
  </xsd:sequence>
  <xsd:anyAttribute processContents="lax"/>
</xsd:complexType>

<xsd:complexType name="ECBoundarySpecExtension">
  <xsd:sequence>
    <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
      namespace="##local"/>
  </xsd:sequence>
  <xsd:anyAttribute processContents="lax"/>
</xsd:complexType>

<xsd:complexType name="ECExcludePatterns">
  <xsd:sequence>
    <xsd:element name="excludePattern" type="xsd:string" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ECFilterSpec">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      A ECFilterSpec specifies what EPCs are to be included in the final
      report. The ECFilterSpec implements a flexible filtering scheme based on
      pattern lists for inclusion and exclusion.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:element name="includePatterns" type="ale:ECIncludePatterns"
      minOccurs="0"/>
    <xsd:element name="excludePatterns" type="ale:ECExcludePatterns"
      minOccurs="0"/>
    <xsd:element name="extension" type="ale:ECFilterSpecExtension"
      minOccurs="0"/>
    <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
      namespace="##other"/>
  </xsd:sequence>
```

```

1401     <xsd:anyAttribute processContents="lax"/>
1402 </xsd:complexType>
1403
1404 <xsd:complexType name="ECFilterSpecExtension">
1405     <xsd:sequence>
1406         <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1407             namespace="##local"/>
1408     </xsd:sequence>
1409     <xsd:anyAttribute processContents="lax"/>
1410 </xsd:complexType>
1411
1412 <xsd:complexType name="ECGroupSpec">
1413     <xsd:sequence>
1414         <xsd:element name="pattern" type="xsd:string"
1415             minOccurs="0" maxOccurs="unbounded"/>
1416     </xsd:sequence>
1417 </xsd:complexType>
1418
1419 <xsd:complexType name="ECIncludePatterns">
1420     <xsd:sequence>
1421         <xsd:element name="includePattern" type="xsd:string" minOccurs="0"
1422             maxOccurs="unbounded"/>
1423     </xsd:sequence>
1424 </xsd:complexType>
1425
1426 <xsd:complexType name="ECLogicalReaders">
1427     <xsd:sequence>
1428         <xsd:element name="logicalReader" type="xsd:string" maxOccurs="unbounded"/>
1429     </xsd:sequence>
1430 </xsd:complexType>
1431
1432 <xsd:complexType name="ECReport">
1433     <xsd:sequence>
1434         <xsd:element name="group" type="ale:ECReportGroup" minOccurs="0"
1435             maxOccurs="unbounded"/>
1436         <xsd:element name="extension" type="ale:ECReportExtension"
1437             minOccurs="0"/>
1438         <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1439             namespace="##other"/>
1440     </xsd:sequence>
1441     <xsd:attribute name="reportName" type="xsd:string" use="required"/>
1442     <xsd:anyAttribute processContents="lax"/>
1443 </xsd:complexType>
1444
1445 <xsd:complexType name="ECReportExtension">
1446     <xsd:sequence>
1447         <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1448             namespace="##local"/>
1449     </xsd:sequence>
1450     <xsd:anyAttribute processContents="lax"/>
1451 </xsd:complexType>
1452
1453 <xsd:complexType name="ECReportList">
1454     <xsd:sequence>
1455         <xsd:element name="report" type="ale:ECReport" minOccurs="0"
1456             maxOccurs="unbounded"/>
1457     </xsd:sequence>
1458 </xsd:complexType>
1459
1460 <xsd:complexType name="ECReportGroup">
1461     <xsd:sequence>
1462         <xsd:element name="groupList" type="ale:ECReportGroupList" minOccurs="0"/>
1463         <xsd:element name="groupCount" type="ale:ECReportGroupCount" minOccurs="0"/>
1464         <xsd:element name="extension" type="ale:ECReportGroupExtension"
1465             minOccurs="0"/>
1466         <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1467             namespace="##other"/>
1468     </xsd:sequence>
1469     <!-- The groupName attribute SHALL be omitted to indicate the default group. -->
1470     <xsd:attribute name="groupName" type="xsd:string" use="optional"/>

```

```

1471     <xsd:anyAttribute processContents="lax"/>
1472 </xsd:complexType>
1473
1474 <xsd:complexType name="ECReportGroupExtension">
1475     <xsd:sequence>
1476         <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1477             namespace="##local"/>
1478     </xsd:sequence>
1479     <xsd:anyAttribute processContents="lax"/>
1480 </xsd:complexType>
1481
1482 <xsd:complexType name="ECReportGroupList">
1483     <xsd:sequence>
1484         <xsd:element name="member" type="ale:ECReportGroupListMember"
1485             minOccurs="0" maxOccurs="unbounded"/>
1486         <xsd:element name="extension" type="ale:ECReportGroupListExtension"
1487             minOccurs="0"/>
1488         <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1489             namespace="##other"/>
1490     </xsd:sequence>
1491 </xsd:complexType>
1492
1493 <xsd:complexType name="ECReportGroupListExtension">
1494     <xsd:sequence>
1495         <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1496             namespace="##local"/>
1497     </xsd:sequence>
1498     <xsd:anyAttribute processContents="lax"/>
1499 </xsd:complexType>
1500
1501 <xsd:complexType name="ECReportGroupListMember">
1502     <xsd:sequence>
1503         <!-- Each of the following four elements SHALL be omitted if null. -->
1504         <xsd:element name="epc" type="epcglobal:EPC" minOccurs="0"/>
1505         <xsd:element name="tag" type="epcglobal:EPC" minOccurs="0"/>
1506         <xsd:element name="rawHex" type="epcglobal:EPC" minOccurs="0"/>
1507         <xsd:element name="rawDecimal" type="epcglobal:EPC" minOccurs="0"/>
1508         <xsd:element name="extension" type="ale:ECReportGroupListMemberExtension"
1509             minOccurs="0"/>
1510         <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1511             namespace="##other"/>
1512     </xsd:sequence>
1513     <xsd:anyAttribute processContents="lax"/>
1514 </xsd:complexType>
1515
1516 <xsd:complexType name="ECReportGroupListMemberExtension">
1517     <xsd:sequence>
1518         <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1519             namespace="##local"/>
1520     </xsd:sequence>
1521     <xsd:anyAttribute processContents="lax"/>
1522 </xsd:complexType>
1523
1524 <xsd:complexType name="ECReportGroupCount">
1525     <xsd:sequence>
1526         <xsd:element name="count" type="xsd:int"/>
1527         <xsd:element name="extension" type="ale:ECReportGroupCountExtension"
1528             minOccurs="0"/>
1529         <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1530             namespace="##other"/>
1531     </xsd:sequence>
1532     <xsd:anyAttribute processContents="lax"/>
1533 </xsd:complexType>
1534
1535 <xsd:complexType name="ECReportGroupCountExtension">
1536     <xsd:sequence>
1537         <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1538             namespace="##local"/>
1539     </xsd:sequence>
1540     <xsd:anyAttribute processContents="lax"/>

```

```

1541 </xsd:complexType>
1542
1543 <xsd:complexType name="ECReportOutputSpec">
1544   <xsd:annotation>
1545     <xsd:documentation xml:lang="en">
1546       ECReportOutputSpec specifies how the final set of EPCs is to be reported
1547       with respect to type.
1548     </xsd:documentation>
1549   </xsd:annotation>
1550   <xsd:sequence>
1551     <xsd:element name="extension" type="ale:ECReportOutputSpecExtension"
1552       minOccurs="0"/>
1553     <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1554       namespace="##other"/>
1555   </xsd:sequence>
1556   <xsd:attribute name="includeEPC" type="xsd:boolean" default="false"/>
1557   <xsd:attribute name="includeTag" type="xsd:boolean" default="false"/>
1558   <xsd:attribute name="includeRawHex" type="xsd:boolean" default="false"/>
1559   <xsd:attribute name="includeRawDecimal" type="xsd:boolean" default="false"/>
1560   <xsd:attribute name="includeCount" type="xsd:boolean" default="false"/>
1561 </xsd:complexType>
1562
1563 <xsd:complexType name="ECReportOutputSpecExtension">
1564   <xsd:sequence>
1565     <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1566       namespace="##local"/>
1567   </xsd:sequence>
1568   <xsd:anyAttribute processContents="lax"/>
1569 </xsd:complexType>
1570
1571
1572 <xsd:complexType name="ECReports">
1573   <xsd:annotation>
1574     <xsd:documentation xml:lang="en">
1575       ECReports is the output from an event cycle. The "meat" of an ECReports
1576       instance is the lists of count report instances and list report
1577       instances, each corresponding to an ECReportSpec instance in the event
1578       cycle's ECSpec. In addition to the reports themselves, ECReports contains
1579       a number of "header" fields that provide useful information about the
1580       event cycle.
1581     </xsd:documentation>
1582   </xsd:annotation>
1583   <xsd:complexContent>
1584     <xsd:extension base="epcglobal:Document">
1585       <xsd:sequence>
1586         <xsd:element name="reports" type="ale:ECReportList"/>
1587         <xsd:element name="extension" type="ale:ECReportsExtension"
1588           minOccurs="0"/>
1589         <xsd:element name="ECSpec" type="ale:ECSpec" minOccurs="0"/>
1590         <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1591           namespace="##other"/>
1592       </xsd:sequence>
1593       <xsd:attribute name="specName" type="xsd:string" use="required"/>
1594       <xsd:attribute name="date" type="xsd:dateTime" use="required"/>
1595       <xsd:attribute name="ALEID" type="xsd:string" use="required"/>
1596       <xsd:attribute name="totalMilliseconds" type="xsd:long" use="required"/>
1597       <xsd:attribute name="terminationCondition"
1598         type="ale:ECTerminationCondition" use="required"/>
1599       <xsd:attribute name="schemaURL" type="xsd:string" use="optional"/>
1600       <xsd:anyAttribute processContents="lax"/>
1601     </xsd:extension>
1602   </xsd:complexContent>
1603 </xsd:complexType>
1604
1605 <xsd:complexType name="ECReportsExtension">
1606   <xsd:sequence>
1607     <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1608       namespace="##local"/>
1609   </xsd:sequence>
1610   <xsd:anyAttribute processContents="lax"/>

```

```

1611 </xsd:complexType>
1612
1613
1614 <xsd:complexType name="ECReportSetSpec">
1615   <xsd:annotation>
1616     <xsd:documentation xml:lang="en">
1617       ECReportSetSpec specifies which set of EPCs is to be considered for
1618       filtering and output.
1619     </xsd:documentation>
1620   </xsd:annotation>
1621   <xsd:attribute name="set" type="ale:ECReportSetEnum"/>
1622 </xsd:complexType>
1623
1624 <xsd:simpleType name="ECReportSetEnum">
1625   <xsd:annotation>
1626     <xsd:documentation xml:lang="en">
1627       ECReportSetEnum is an enumerated type denoting what set of EPCs is to be
1628       considered for filtering and output: all EPCs read in the current event
1629       cycle, additions from the previous event cycle, or deletions from the
1630       previous event cycle.
1631     </xsd:documentation>
1632   </xsd:annotation>
1633   <xsd:restriction base="xsd:NCName">
1634     <xsd:enumeration value="CURRENT"/>
1635     <xsd:enumeration value="ADDITIONS"/>
1636     <xsd:enumeration value="DELETIONS"/>
1637   </xsd:restriction>
1638 </xsd:simpleType>
1639
1640 <xsd:complexType name="ECReportSpec">
1641   <xsd:annotation>
1642     <xsd:documentation xml:lang="en">
1643       A ReportSpec specifies one report to be returned from executing an event
1644       cycle. An ECSpec may contain one or more ECReportSpec instances.
1645     </xsd:documentation>
1646   </xsd:annotation>
1647   <xsd:sequence>
1648     <xsd:element name="reportSet" type="ale:ECReportSetSpec"/>
1649     <xsd:element name="filterSpec" type="ale:ECFilterSpec" minOccurs="0"/>
1650     <xsd:element name="groupSpec" type="ale:ECGroupSpec" minOccurs="0"/>
1651     <xsd:element name="output" type="ale:ECReportOutputSpec"/>
1652     <xsd:element name="extension" type="ale:ECReportSpecExtension"
1653       minOccurs="0"/>
1654     <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1655       namespace="##other"/>
1656   </xsd:sequence>
1657   <xsd:attribute name="reportName" type="xsd:string" use="required"/>
1658   <xsd:attribute name="reportIfEmpty" type="xsd:boolean" default="false"/>
1659   <xsd:attribute name="reportOnlyOnChange" type="xsd:boolean" default="false"/>
1660   <xsd:anyAttribute processContents="lax"/>
1661 </xsd:complexType>
1662
1663 <xsd:complexType name="ECReportSpecExtension">
1664   <xsd:sequence>
1665     <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1666       namespace="##local"/>
1667   </xsd:sequence>
1668   <xsd:anyAttribute processContents="lax"/>
1669 </xsd:complexType>
1670
1671
1672 <xsd:complexType name="ECReportSpecs">
1673   <xsd:sequence>
1674     <xsd:element name="reportSpec" type="ale:ECReportSpec"
1675       maxOccurs="unbounded"/>
1676   </xsd:sequence>
1677 </xsd:complexType>
1678
1679 <xsd:complexType name="ECSpec">
1680   <xsd:annotation>

```



```

1681         <xsd:documentation xml:lang="en">
1682             An ECSpec describes an event cycle and one or more reports that are to
1683             be generated from it. It contains a list of logical readers whose reader
1684             cycles are to be included in the event cycle, a specification of read
1685             cycle timing, a specification of how the boundaries of event cycles are
1686             to be determined, and list of specifications each of which describes a
1687             report to be generated from this event cycle.
1688         </xsd:documentation>
1689     </xsd:annotation>
1690     <xsd:complexContent>
1691         <xsd:extension base="epcglobal:Document">
1692             <xsd:sequence>
1693                 <xsd:element name="logicalReaders" type="ale:ECLogicalReaders"/>
1694                 <xsd:element name="boundarySpec" type="ale:ECBoundarySpec"/>
1695                 <xsd:element name="reportSpecs" type="ale:ECReportSpecs"/>
1696                 <xsd:element name="extension" type="ale:ECSpecExtension"
1697                     minOccurs="0"/>
1698                 <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
1699                     namespace="##other"/>
1700             </xsd:sequence>
1701             <xsd:attribute name="includeSpecInReports" type="xsd:boolean"
1702                 default="false"/>
1703             <xsd:anyAttribute processContents="lax"/>
1704         </xsd:extension>
1705     </xsd:complexContent>
1706 </xsd:complexType>
1707
1708 <xsd:complexType name="ECSpecExtension">
1709     <xsd:sequence>
1710         <xsd:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
1711             namespace="##local"/>
1712     </xsd:sequence>
1713     <xsd:anyAttribute processContents="lax"/>
1714 </xsd:complexType>
1715
1716 <xsd:simpleType name="ECTerminationCondition">
1717     <xsd:restriction base="xsd:NCName">
1718         <xsd:enumeration value="TRIGGER"/>
1719         <xsd:enumeration value="DURATION"/>
1720         <xsd:enumeration value="STABLE_SET"/>
1721         <xsd:enumeration value="UNREQUEST"/>
1722     </xsd:restriction>
1723 </xsd:simpleType>
1724
1725 <xsd:complexType name="ECTime">
1726     <xsd:annotation>
1727         <xsd:documentation xml:lang="en">
1728             An ECTime specifies a time duration in physical units.
1729         </xsd:documentation>
1730     </xsd:annotation>
1731     <xsd:simpleContent>
1732         <xsd:extension base="xsd:long">
1733             <xsd:attribute name="unit" type="ale:ECTimeUnit"/>
1734         </xsd:extension>
1735     </xsd:simpleContent>
1736 </xsd:complexType>
1737
1738 <xsd:simpleType name="ECTimeUnit">
1739     <xsd:annotation>
1740         <xsd:documentation xml:lang="en">
1741             ECTimeUnit represents the supported physical time unit: millisecond
1742         </xsd:documentation>
1743     </xsd:annotation>
1744     <xsd:restriction base="xsd:NCName">
1745         <xsd:enumeration value="MS"/>
1746     </xsd:restriction>
1747 </xsd:simpleType>
1748
1749 <xsd:complexType name="ECTrigger">

```

```

1751     <xsd:annotation>
1752       <xsd:documentation xml:lang="en">
1753         A trigger is a URI that is used to specify a start or stop trigger for
1754         an event cycle.
1755       </xsd:documentation>
1756     </xsd:annotation>
1757     <xsd:simpleContent>
1758       <xsd:extension base="xsd:string"/>
1759     </xsd:simpleContent>
1760   </xsd:complexType>
1761 </xsd:schema>

```

10.3 ECSpec – Example (non-normative)

Here is an example ECSpec rendered into XML [XML1.0]:

```

1764 <?xml version="1.0" encoding="UTF-8"?>
1765 <ale:ECSpec xmlns:ale="urn:epcglobal:ale:xsd:1"
1766   xmlns:epcglobal="urn:epcglobal:xsd:1"
1767   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1768   xsi:schemaLocation="urn:epcglobal:ale:xsd:1 Ale.xsd"
1769   schemaVersion="1.0"
1770   creationDate="2003-08-06T10:54:06.444-05:00">
1771   <logicalReaders>
1772     <logicalReader>dock_1a</logicalReader>
1773     <logicalReader>dock_1b</logicalReader>
1774   </logicalReaders>
1775   <boundarySpec>
1776     <startTrigger>http://sample.com/trigger1</startTrigger>
1777     <repeatPeriod unit="MS">20000</repeatPeriod>
1778     <stopTrigger>http://sample.com/trigger2</stopTrigger>
1779     <duration unit="MS">3000</duration>
1780   </boundarySpec>
1781   <reportSpecs>
1782     <reportSpec reportName="report1">
1783       <reportSet set="CURRENT"/>
1784       <output includeTag="true"/>
1785     </reportSpec>
1786     <reportSpec reportName="report2">
1787       <reportSet set="ADDITIONS"/>
1788       <output includeCount="true"/>
1789     </reportSpec>
1790     <reportSpec reportName="report3">
1791       <reportSet set="DELETIONS"/>
1792       <groupSpec>
1793         <pattern>urn:epc:pat:sgtin-64:X.X.X.*</pattern>
1794       </groupSpec>
1795       <output includeCount="true"/>
1796     </reportSpec>
1797   </reportSpecs>
1798 </ale:ECSpec>

```

10.4 ECREports – Example (non-normative)

Here is an example ECREports rendered into XML [XML1.0]:

```

1801 <?xml version="1.0" encoding="UTF-8"?>
1802 <ale:ECReports xmlns:ale="urn:epcglobal:ale:xsd:1"
1803   xmlns:epcglobal="urn:epcglobal:xsd:1"
1804   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1805   xsi:schemaLocation="urn:epcglobal:ale:xsd:1 Ale.xsd"
1806   schemaVersion="1.0"
1807   creationDate="2003-08-06T10:54:06.444-05:00"
1808   specName="EventCycle1"
1809   date="2003-08-06T10:54:06.444-05:00"
1810   ALEID="Edge34"
1811   totalMilliseconds="3034"
1812   terminationCondition="DURATION">

```

```

1813     <reports>
1814       <report reportName="report1">
1815         <group>
1816           <groupList>
1817             <member><tag>urn:epc:tag:gid-96:10.50.1000</tag></member>
1818             <member><tag>urn:epc:tag:gid-96:10.50.1001</tag></member>
1819           </groupList>
1820         </group>
1821       </report>
1822       <report reportName="report2">
1823         <group><groupCount><count>6847</count></groupCount></group>
1824       </report>
1825       <report reportName="report3">
1826         <group name="urn:epc:pat:sgtin-64:3.0037000.12345.*">
1827           <groupCount><count>2</count></groupCount>
1828         </group>
1829         <group name="urn:epc:pat:sgtin-64:3.0037000.55555.*">
1830           <groupCount><count>3</count></groupCount>
1831         </group>
1832         <group>
1833           <groupCount><count>6842</count></groupCount>
1834         </group>
1835       </report>
1836     </reports>
1837 </ale:ECReports>

```

11 SOAP Binding for ALE API

11.1 SOAP Binding

The following is a Web Service Definition Language (WSDL) 1.1 [WSDL1.1] specification defining the standard SOAP binding of the ALE API. This SOAP binding is compliant with the WS-i Basic Profile Version 1.0 [WSI].

```

1843 <?xml version="1.0" encoding="UTF-8"?>
1844
1845 <!-- ALESERVICE DEFINITIONS -->
1846 <wsdl:definitions
1847   targetNamespace="urn:epcglobal:ale:wsdl:1"
1848   xmlns="http://schemas.xmlsoap.org/wsdl/"
1849   xmlns:impl="urn:epcglobal:ale:wsdl:1"
1850   xmlns:ale="urn:epcglobal:ale:xsd:1"
1851   xmlns:epcglobal="urn:epcglobal:xsd:1"
1852   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
1853   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
1854   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
1855   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
1856
1857   <wsdl:documentation>
1858     <epcglobal:copyright>Copyright (C) 2005, 2004 EPCglobal Inc., All Rights
1859     Reserved.</epcglobal:copyright>
1860     <epcglobal:disclaimer>EPCglobal Inc., its members, officers, directors, employees,
1861     or agents shall not be liable for any injury, loss, damages, financial or otherwise,
1862     arising from, related to, or caused by the use of this document. The use of said
1863     document shall constitute your express consent to the foregoing
1864     exculpation.</epcglobal:disclaimer>
1865     <epcglobal:specification></epcglobal:specification>
1866
1867     This WSDL document describes the types, messages, operations, and
1868     bindings for the ALEService.
1869   </wsdl:documentation>
1870
1871   <!-- ALESERVICE TYPES -->
1872   <wsdl:types>
1873     <xsd:schema targetNamespace="urn:epcglobal:ale:wsdl:1"
1874               xmlns:impl="urn:epcglobal:ale:wsdl:1"

```

```

1875         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
1876 <xsd:import namespace="urn:epcglobal:ale:xsd:1"
1877           schemaLocation="./ALE.xsd"/>
1878
1879 <!-- ALESERVICE MESSAGE WRAPPERS -->
1880 <xsd:element name="Define" type="impl:Define"/>
1881 <xsd:complexType name="Define">
1882   <xsd:sequence>
1883     <xsd:element name="specName" type="xsd:string"/>
1884     <xsd:element name="spec" type="ale:ECSpec"/>
1885   </xsd:sequence>
1886 </xsd:complexType>
1887
1888 <xsd:element name="Undefine" type="impl:Undefine"/>
1889 <xsd:complexType name="Undefine">
1890   <xsd:sequence>
1891     <xsd:element name="specName" type="xsd:string"/>
1892   </xsd:sequence>
1893 </xsd:complexType>
1894
1895 <xsd:element name="GetECSpec" type="impl:GetECSpec"/>
1896 <xsd:complexType name="GetECSpec">
1897   <xsd:sequence>
1898     <xsd:element name="specName" type="xsd:string"/>
1899   </xsd:sequence>
1900 </xsd:complexType>
1901 <xsd:element name="GetECSpecResult" type="ale:ECSpec"/>
1902
1903 <xsd:element name="GetECSpecNames" type="impl:EmptyParms"/>
1904 <xsd:element name="GetECSpecNamesResult" type="impl:ArrayOfString"/>
1905
1906 <xsd:element name="Subscribe" type="impl:Subscribe"/>
1907 <xsd:complexType name="Subscribe">
1908   <xsd:sequence>
1909     <xsd:element name="specName" type="xsd:string"/>
1910     <xsd:element name="notificationURI" type="xsd:string"/>
1911   </xsd:sequence>
1912 </xsd:complexType>
1913
1914 <xsd:element name="Unsubscribe" type="impl:Unsubscribe"/>
1915 <xsd:complexType name="Unsubscribe">
1916   <xsd:sequence>
1917     <xsd:element name="specName" type="xsd:string"/>
1918     <xsd:element name="notificationURI" type="xsd:string"/>
1919   </xsd:sequence>
1920 </xsd:complexType>
1921
1922 <xsd:element name="Poll" type="impl:Poll"/>
1923 <xsd:complexType name="Poll">
1924   <xsd:sequence>
1925     <xsd:element name="specName" type="xsd:string"/>
1926   </xsd:sequence>
1927 </xsd:complexType>
1928 <xsd:element name="PollResult" type="ale:ECReports"/>
1929
1930 <xsd:element name="Immediate" type="impl:Immediate"/>
1931 <xsd:complexType name="Immediate">
1932   <xsd:sequence>
1933     <xsd:element name="spec" type="ale:ECSpec"/>
1934   </xsd:sequence>
1935 </xsd:complexType>
1936 <xsd:element name="ImmediateResult" type="ale:ECReports"/>
1937
1938 <xsd:element name="GetSubscribers" type="impl:GetSubscribers"/>
1939 <xsd:complexType name="GetSubscribers">
1940   <xsd:sequence>
1941     <xsd:element name="specName" type="xsd:string"/>
1942   </xsd:sequence>
1943 </xsd:complexType>
1944 <xsd:element name="GetSubscribersResult" type="impl:ArrayOfString"/>

```

```

1945
1946 <xsd:element name="GetStandardVersion" type="impl:EmptyParms"/>
1947 <xsd:element name="GetStandardVersionResult" type="xsd:string"/>
1948
1949 <xsd:element name="GetVendorVersion" type="impl:EmptyParms"/>
1950 <xsd:element name="GetVendorVersionResult" type="xsd:string"/>
1951
1952 <xsd:element name="VoidHolder" type="impl:VoidHolder"/>
1953 <xsd:complexType name="VoidHolder">
1954   <xsd:sequence>
1955   </xsd:sequence>
1956 </xsd:complexType>
1957
1958 <xsd:complexType name="EmptyParms"/>
1959
1960 <xsd:complexType name="ArrayOfString">
1961   <xsd:sequence>
1962     <xsd:element name="string" type="xsd:string" minOccurs="0"
1963       maxOccurs="unbounded"/>
1964   </xsd:sequence>
1965 </xsd:complexType>
1966
1967 <!-- ALE EXCEPTIONS -->
1968 <xsd:element name="ALEException" type="impl:ALEException"/>
1969 <xsd:complexType name="ALEException">
1970   <xsd:sequence>
1971     <xsd:element name="reason" type="xsd:string"/>
1972   </xsd:sequence>
1973 </xsd:complexType>
1974
1975 <xsd:element name="SecurityException"
1976   type="impl:SecurityException"/>
1977 <xsd:complexType name="SecurityException">
1978   <xsd:complexContent>
1979     <xsd:extension base="impl:ALEException">
1980       <xsd:sequence/>
1981     </xsd:extension>
1982   </xsd:complexContent>
1983 </xsd:complexType>
1984
1985 <xsd:element name="DuplicateNameException"
1986   type="impl:DuplicateNameException"/>
1987 <xsd:complexType name="DuplicateNameException">
1988   <xsd:complexContent>
1989     <xsd:extension base="impl:ALEException">
1990       <xsd:sequence/>
1991     </xsd:extension>
1992   </xsd:complexContent>
1993 </xsd:complexType>
1994
1995 <xsd:element name="ECSpecValidationException"
1996   type="impl:ECSpecValidationException"/>
1997 <xsd:complexType name="ECSpecValidationException">
1998   <xsd:complexContent>
1999     <xsd:extension base="impl:ALEException">
2000       <xsd:sequence/>
2001     </xsd:extension>
2002   </xsd:complexContent>
2003 </xsd:complexType>
2004
2005 <xsd:element name="InvalidURIException" type="impl:InvalidURIException"/>
2006 <xsd:complexType name="InvalidURIException">
2007   <xsd:complexContent>
2008     <xsd:extension base="impl:ALEException">
2009       <xsd:sequence/>
2010     </xsd:extension>
2011   </xsd:complexContent>
2012 </xsd:complexType>
2013
2014 <xsd:element name="NoSuchNameException" type="impl:NoSuchNameException"/>

```

```

2015     <xsd:complexType name="NoSuchNameException">
2016         <xsd:complexContent>
2017             <xsd:extension base="impl:ALEException">
2018                 <xsd:sequence/>
2019             </xsd:extension>
2020         </xsd:complexContent>
2021     </xsd:complexType>
2022
2023     <xsd:element name="NoSuchSubscriberException"
2024                 type="impl:NoSuchSubscriberException"/>
2025 <xsd:complexType name="NoSuchSubscriberException">
2026     <xsd:complexContent>
2027         <xsd:extension base="impl:ALEException">
2028             <xsd:sequence/>
2029         </xsd:extension>
2030     </xsd:complexContent>
2031 </xsd:complexType>
2032
2033     <xsd:element name="DuplicateSubscriptionException"
2034                 type="impl:DuplicateSubscriptionException"/>
2035 <xsd:complexType name="DuplicateSubscriptionException">
2036     <xsd:complexContent>
2037         <xsd:extension base="impl:ALEException">
2038             <xsd:sequence/>
2039         </xsd:extension>
2040     </xsd:complexContent>
2041 </xsd:complexType>
2042
2043     <xsd:element name="ImplementationException"
2044                 type="impl:ImplementationException"/>
2045 <xsd:complexType name="ImplementationException">
2046     <xsd:complexContent>
2047         <xsd:extension base="impl:ALEException">
2048             <xsd:sequence>
2049                 <xsd:element name="severity"
2050                             type="impl:ImplementationExceptionSeverity"/>
2051             </xsd:sequence>
2052         </xsd:extension>
2053     </xsd:complexContent>
2054 </xsd:complexType>
2055
2056 <xsd:simpleType name="ImplementationExceptionSeverity">
2057     <xsd:restriction base="xsd:NCName">
2058         <xsd:enumeration value="ERROR"/>
2059         <xsd:enumeration value="SEVERE"/>
2060     </xsd:restriction>
2061 </xsd:simpleType>
2062
2063 </xsd:schema>
2064 </wsdl:types>
2065
2066 <!-- ALESERVICE MESSAGES -->
2067 <wsdl:message name="defineRequest">
2068     <wsdl:part name="parms" element="impl:Define"/>
2069 </wsdl:message>
2070 <wsdl:message name="defineResponse">
2071     <wsdl:part name="defineReturn" element="impl:VoidHolder"/>
2072 </wsdl:message>
2073
2074 <wsdl:message name="undefineRequest">
2075     <wsdl:part name="parms" element="impl:Undefine"/>
2076 </wsdl:message>
2077 <wsdl:message name="undefineResponse">
2078     <wsdl:part name="undefineReturn" element="impl:VoidHolder"/>
2079 </wsdl:message>
2080
2081 <wsdl:message name="getECSpecRequest">
2082     <wsdl:part name="parms" element="impl:GetECSpec"/>
2083 </wsdl:message>
2084 <wsdl:message name="getECSpecResponse">

```

```

2085     <wsdl:part name="getECSpecReturn" element="impl:GetECSpecResult"/>
2086 </wsdl:message>
2087
2088 <wsdl:message name="getECSpecNamesRequest">
2089     <wsdl:part name="parms" element="impl:GetECSpecNames"/>
2090 </wsdl:message>
2091 <wsdl:message name="getECSpecNamesResponse">
2092     <wsdl:part name="getECSpecNamesReturn" element="impl:GetECSpecNamesResult"/>
2093 </wsdl:message>
2094
2095 <wsdl:message name="subscribeRequest">
2096     <wsdl:part name="parms" element="impl:Subscribe"/>
2097 </wsdl:message>
2098 <wsdl:message name="subscribeResponse">
2099     <wsdl:part name="subscribeReturn" element="impl:VoidHolder"/>
2100 </wsdl:message>
2101
2102 <wsdl:message name="unsubscribeRequest">
2103     <wsdl:part name="parms" element="impl:Unsubscribe"/>
2104 </wsdl:message>
2105 <wsdl:message name="unsubscribeResponse">
2106     <wsdl:part name="unsubscribeReturn" element="impl:VoidHolder"/>
2107 </wsdl:message>
2108
2109 <wsdl:message name="pollRequest">
2110     <wsdl:part name="parms" element="impl:Poll"/>
2111 </wsdl:message>
2112 <wsdl:message name="pollResponse">
2113     <wsdl:part name="pollReturn" element="impl:PollResult"/>
2114 </wsdl:message>
2115
2116 <wsdl:message name="immediateRequest">
2117     <wsdl:part name="parms" element="impl:Immediate"/>
2118 </wsdl:message>
2119 <wsdl:message name="immediateResponse">
2120     <wsdl:part name="immediateReturn" element="impl:ImmediateResult"/>
2121 </wsdl:message>
2122
2123 <wsdl:message name="getSubscribersRequest">
2124     <wsdl:part name="parms" element="impl:GetSubscribers"/>
2125 </wsdl:message>
2126 <wsdl:message name="getSubscribersResponse">
2127     <wsdl:part name="getSubscribersReturn" element="impl:GetSubscribersResult"/>
2128 </wsdl:message>
2129
2130 <wsdl:message name="getStandardVersionRequest">
2131     <wsdl:part name="parms" element="impl:GetStandardVersion"/>
2132 </wsdl:message>
2133 <wsdl:message name="getStandardVersionResponse">
2134     <wsdl:part name="getStandardVersionReturn"
2135 element="impl:GetStandardVersionResult"/>
2136 </wsdl:message>
2137
2138
2139 <wsdl:message name="getVendorVersionRequest">
2140     <wsdl:part name="parms" element="impl:GetVendorVersion"/>
2141 </wsdl:message>
2142 <wsdl:message name="getVendorVersionResponse">
2143     <wsdl:part name="getVendorVersionReturn" element="impl:GetVendorVersionResult"/>
2144 </wsdl:message>
2145
2146 <!-- ALESERVICE FAULT EXCEPTIONS -->
2147 <wsdl:message name="DuplicateNameExceptionResponse">
2148     <wsdl:part name="fault" element="impl:DuplicateNameException"/>
2149 </wsdl:message>
2150 <wsdl:message name="ECSpecValidationExceptionResponse">
2151     <wsdl:part name="fault" element="impl:ECSpecValidationException"/>
2152 </wsdl:message>
2153 <wsdl:message name="InvalidURIExceptionResponse">
2154     <wsdl:part name="fault" element="impl:InvalidURIException"/>

```



```

2155 </wsdl:message>
2156 <wsdl:message name="NoSuchNameExceptionResponse">
2157   <wsdl:part name="fault" element="impl:NoSuchNameException"/>
2158 </wsdl:message>
2159 <wsdl:message name="NoSuchSubscriberExceptionResponse">
2160   <wsdl:part name="fault" element="impl:NoSuchSubscriberException"/>
2161 </wsdl:message>
2162 <wsdl:message name="DuplicateSubscriptionExceptionResponse">
2163   <wsdl:part name="fault" element="impl:DuplicateSubscriptionException"/>
2164 </wsdl:message>
2165 <wsdl:message name="ImplementationExceptionResponse">
2166   <wsdl:part name="fault" element="impl:ImplementationException"/>
2167 </wsdl:message>
2168 <wsdl:message name="SecurityExceptionResponse">
2169   <wsdl:part name="fault" element="impl:SecurityException"/>
2170 </wsdl:message>
2171
2172 <!-- ALESERVICE PORTTYPE -->
2173 <wsdl:portType name="ALEServicePortType">
2174   <wsdl:operation name="define">
2175     <wsdl:input message="impl:defineRequest" name="defineRequest"/>
2176     <wsdl:output message="impl:defineResponse" name="defineResponse"/>
2177     <wsdl:fault message="impl:DuplicateNameExceptionResponse"
2178       name="DuplicateNameExceptionFault"/>
2179     <wsdl:fault message="impl:ECSpecValidationExceptionResponse"
2180       name="ECSpecValidationExceptionFault"/>
2181     <wsdl:fault message="impl:SecurityExceptionResponse"
2182       name="SecurityExceptionFault"/>
2183     <wsdl:fault message="impl:ImplementationExceptionResponse"
2184       name="ImplementationExceptionFault"/>
2185   </wsdl:operation>
2186
2187   <wsdl:operation name="undefine">
2188     <wsdl:input message="impl:undefineRequest" name="undefineRequest"/>
2189     <wsdl:output message="impl:undefineResponse" name="undefineResponse"/>
2190     <wsdl:fault message="impl:NoSuchNameExceptionResponse"
2191       name="NoSuchNameExceptionFault"/>
2192     <wsdl:fault message="impl:SecurityExceptionResponse"
2193       name="SecurityExceptionFault"/>
2194     <wsdl:fault message="impl:ImplementationExceptionResponse"
2195       name="ImplementationExceptionFault"/>
2196   </wsdl:operation>
2197
2198   <wsdl:operation name="getECSpec">
2199     <wsdl:input message="impl:getECSpecRequest" name="getECSpecRequest"/>
2200     <wsdl:output message="impl:getECSpecResponse" name="getECSpecResponse"/>
2201     <wsdl:fault message="impl:NoSuchNameExceptionResponse"
2202       name="NoSuchNameExceptionFault"/>
2203     <wsdl:fault message="impl:SecurityExceptionResponse"
2204       name="SecurityExceptionFault"/>
2205     <wsdl:fault message="impl:ImplementationExceptionResponse"
2206       name="ImplementationExceptionFault"/>
2207   </wsdl:operation>
2208
2209   <wsdl:operation name="getECSpecNames">
2210     <wsdl:input message="impl:getECSpecNamesRequest"
2211       name="getECSpecNamesRequest"/>
2212     <wsdl:output message="impl:getECSpecNamesResponse"
2213       name="getECSpecNamesResponse"/>
2214     <wsdl:fault message="impl:SecurityExceptionResponse"
2215       name="SecurityExceptionFault"/>
2216     <wsdl:fault message="impl:ImplementationExceptionResponse"
2217       name="ImplementationExceptionFault"/>
2218   </wsdl:operation>
2219
2220   <wsdl:operation name="subscribe">
2221     <wsdl:input message="impl:subscribeRequest" name="subscribeRequest"/>
2222     <wsdl:output message="impl:subscribeResponse" name="subscribeResponse"/>
2223     <wsdl:fault message="impl:NoSuchNameExceptionResponse"
2224       name="NoSuchNameExceptionFault"/>

```

```

2225     <wsdl:fault message="impl:InvalidURIExceptionResponse"
2226               name="InvalidURIExceptionFault"/>
2227     <wsdl:fault message="impl:DuplicateSubscriptionExceptionResponse"
2228               name="DuplicateSubscriptionExceptionFault"/>
2229     <wsdl:fault message="impl:SecurityExceptionResponse"
2230               name="SecurityExceptionFault"/>
2231     <wsdl:fault message="impl:ImplementationExceptionResponse"
2232               name="ImplementationExceptionFault"/>
2233   </wsdl:operation>
2234
2235   <wsdl:operation name="unsubscribe">
2236     <wsdl:input message="impl:unsubscribeRequest" name="unsubscribeRequest"/>
2237     <wsdl:output message="impl:unsubscribeResponse" name="unsubscribeResponse"/>
2238     <wsdl:fault message="impl:NoSuchNameExceptionResponse"
2239               name="NoSuchNameExceptionFault"/>
2240     <wsdl:fault message="impl:NoSuchSubscriberExceptionResponse"
2241               name="NoSuchSubscriberExceptionFault"/>
2242     <wsdl:fault message="impl:InvalidURIExceptionResponse"
2243               name="InvalidURIExceptionFault"/>
2244     <wsdl:fault message="impl:SecurityExceptionResponse"
2245               name="SecurityExceptionFault"/>
2246     <wsdl:fault message="impl:ImplementationExceptionResponse"
2247               name="ImplementationExceptionFault"/>
2248   </wsdl:operation>
2249
2250   <wsdl:operation name="poll">
2251     <wsdl:input message="impl:pollRequest" name="pollRequest"/>
2252     <wsdl:output message="impl:pollResponse" name="pollResponse"/>
2253     <wsdl:fault message="impl:NoSuchNameExceptionResponse"
2254               name="NoSuchNameExceptionFault"/>
2255     <wsdl:fault message="impl:SecurityExceptionResponse"
2256               name="SecurityExceptionFault"/>
2257     <wsdl:fault message="impl:ImplementationExceptionResponse"
2258               name="ImplementationExceptionFault"/>
2259   </wsdl:operation>
2260
2261   <wsdl:operation name="immediate">
2262     <wsdl:input message="impl:immediateRequest" name="immediateRequest"/>
2263     <wsdl:output message="impl:immediateResponse" name="immediateResponse"/>
2264     <wsdl:fault message="impl:ECSpecValidationExceptionResponse"
2265               name="ECSpecValidationExceptionFault"/>
2266     <wsdl:fault message="impl:SecurityExceptionResponse"
2267               name="SecurityExceptionFault"/>
2268     <wsdl:fault message="impl:ImplementationExceptionResponse"
2269               name="ImplementationExceptionFault"/>
2270   </wsdl:operation>
2271
2272   <wsdl:operation name="getSubscribers">
2273     <wsdl:input message="impl:getSubscribersRequest"
2274               name="getSubscribersRequest"/>
2275     <wsdl:output message="impl:getSubscribersResponse"
2276               name="getSubscribersResponse"/>
2277     <wsdl:fault message="impl:NoSuchNameExceptionResponse"
2278               name="NoSuchNameExceptionFault"/>
2279     <wsdl:fault message="impl:SecurityExceptionResponse"
2280               name="SecurityExceptionFault"/>
2281     <wsdl:fault message="impl:ImplementationExceptionResponse"
2282               name="ImplementationExceptionFault"/>
2283   </wsdl:operation>
2284
2285   <wsdl:operation name="getStandardVersion">
2286     <wsdl:input message="impl:getStandardVersionRequest"
2287               name="getStandardVersionRequest"/>
2288     <wsdl:output message="impl:getStandardVersionResponse"
2289               name="getStandardVersionResponse"/>
2290     <wsdl:fault message="impl:ImplementationExceptionResponse"
2291               name="ImplementationExceptionFault"/>
2292   </wsdl:operation>
2293
2294   <wsdl:operation name="getVendorVersion">

```

```

2295         <wsdl:input message="impl:getVendorVersionRequest"
2296 name="getVendorVersionRequest"/>
2297         <wsdl:output message="impl:getVendorVersionResponse"
2298 name="getVendorVersionResponse"/>
2299         <wsdl:fault message="impl:ImplementationExceptionResponse"
2300 name="ImplementationExceptionFault"/>
2301     </wsdl:operation>
2302     </wsdl:portType>
2303
2304     <!-- ALESERVICE BINDING -->
2305     <wsdl:binding name="ALEServiceBinding" type="impl:ALEServicePortType">
2306         <wsdlsoap:binding style="document"
2307             transport="http://schemas.xmlsoap.org/soap/http"/>
2308         <wsdl:operation name="define">
2309             <wsdlsoap:operation soapAction=""/>
2310             <wsdl:input name="defineRequest">
2311                 <wsdlsoap:body
2312                     use="literal"/>
2313             </wsdl:input>
2314             <wsdl:output name="defineResponse">
2315                 <wsdlsoap:body
2316                     use="literal"/>
2317             </wsdl:output>
2318             <wsdl:fault name="DuplicateNameExceptionFault">
2319                 <wsdlsoap:fault
2320                     name="DuplicateNameExceptionFault"
2321                     use="literal"/>
2322             </wsdl:fault>
2323             <wsdl:fault name="ECSpecValidationExceptionFault">
2324                 <wsdlsoap:fault
2325                     name="ECSpecValidationExceptionFault"
2326                     use="literal"/>
2327             </wsdl:fault>
2328             <wsdl:fault name="SecurityExceptionFault">
2329                 <wsdlsoap:fault
2330                     name="SecurityExceptionFault"
2331                     use="literal"/>
2332             </wsdl:fault>
2333             <wsdl:fault name="ImplementationExceptionFault">
2334                 <wsdlsoap:fault
2335                     name="ImplementationExceptionFault"
2336                     use="literal"/>
2337             </wsdl:fault>
2338         </wsdl:operation>
2339
2340         <wsdl:operation name="undefine">
2341             <wsdlsoap:operation soapAction=""/>
2342             <wsdl:input name="undefineRequest">
2343                 <wsdlsoap:body
2344                     use="literal"/>
2345             </wsdl:input>
2346             <wsdl:output name="undefineResponse">
2347                 <wsdlsoap:body
2348                     use="literal"/>
2349             </wsdl:output>
2350             <wsdl:fault name="NoSuchNameExceptionFault">
2351                 <wsdlsoap:fault
2352                     name="NoSuchNameExceptionFault"
2353                     use="literal"/>
2354             </wsdl:fault>
2355             <wsdl:fault name="SecurityExceptionFault">
2356                 <wsdlsoap:fault
2357                     name="SecurityExceptionFault"
2358                     use="literal"/>
2359             </wsdl:fault>
2360             <wsdl:fault name="ImplementationExceptionFault">
2361                 <wsdlsoap:fault
2362                     name="ImplementationExceptionFault"
2363                     use="literal"/>
2364             </wsdl:fault>
2365         </wsdl:operation>

```

```

2365
2366 <wsdl:operation name="getECSpec">
2367   <wsdlsoap:operation soapAction=""/>
2368   <wsdl:input name="getECSpecRequest">
2369     <wsdlsoap:body
2370       use="literal"/>
2371   </wsdl:input>
2372   <wsdl:output name="getECSpecResponse">
2373     <wsdlsoap:body
2374       use="literal"/>
2375   </wsdl:output>
2376   <wsdl:fault name="NoSuchNameExceptionFault">
2377     <wsdlsoap:fault
2378       name="NoSuchNameExceptionFault"
2379       use="literal"/>
2380   </wsdl:fault>
2381   <wsdl:fault name="SecurityExceptionFault">
2382     <wsdlsoap:fault
2383       name="SecurityExceptionFault"
2384       use="literal"/>
2385   </wsdl:fault>
2386   <wsdl:fault name="ImplementationExceptionFault">
2387     <wsdlsoap:fault
2388       name="ImplementationExceptionFault"
2389       use="literal"/>
2390   </wsdl:fault>
2391 </wsdl:operation>
2392
2393 <wsdl:operation name="getECSpecNames">
2394   <wsdlsoap:operation soapAction=""/>
2395   <wsdl:input name="getECSpecNamesRequest">
2396     <wsdlsoap:body
2397       use="literal"/>
2398   </wsdl:input>
2399   <wsdl:output name="getECSpecNamesResponse">
2400     <wsdlsoap:body
2401       use="literal"/>
2402   </wsdl:output>
2403   <wsdl:fault name="SecurityExceptionFault">
2404     <wsdlsoap:fault
2405       name="SecurityExceptionFault"
2406       use="literal"/>
2407   </wsdl:fault>
2408   <wsdl:fault name="ImplementationExceptionFault">
2409     <wsdlsoap:fault
2410       name="ImplementationExceptionFault"
2411       use="literal"/>
2412   </wsdl:fault>
2413 </wsdl:operation>
2414
2415 <wsdl:operation name="subscribe">
2416   <wsdlsoap:operation soapAction=""/>
2417   <wsdl:input name="subscribeRequest">
2418     <wsdlsoap:body
2419       use="literal"/>
2420   </wsdl:input>
2421   <wsdl:output name="subscribeResponse">
2422     <wsdlsoap:body
2423       use="literal"/>
2424   </wsdl:output>
2425   <wsdl:fault name="NoSuchNameExceptionFault">
2426     <wsdlsoap:fault
2427       name="NoSuchNameExceptionFault"
2428       use="literal"/>
2429   </wsdl:fault>
2430   <wsdl:fault name="InvalidURIExceptionFault">
2431     <wsdlsoap:fault
2432       name="InvalidURIExceptionFault"
2433       use="literal"/>
2434   </wsdl:fault>

```

```

2435     <wsdl:fault name="DuplicateSubscriptionExceptionFault">
2436         <wsdlsoap:fault
2437             name="DuplicateSubscriptionExceptionFault"
2438             use="literal"/>
2439     </wsdl:fault>
2440     <wsdl:fault name="SecurityExceptionFault">
2441         <wsdlsoap:fault
2442             name="SecurityExceptionFault"
2443             use="literal"/>
2444     </wsdl:fault>
2445     <wsdl:fault name="ImplementationExceptionFault">
2446         <wsdlsoap:fault
2447             name="ImplementationExceptionFault"
2448             use="literal"/>
2449     </wsdl:fault>
2450 </wsdl:operation>
2451
2452 <wsdl:operation name="unsubscribe">
2453     <wsdlsoap:operation soapAction=""/>
2454     <wsdl:input name="unsubscribeRequest">
2455         <wsdlsoap:body
2456             use="literal"/>
2457     </wsdl:input>
2458     <wsdl:output name="unsubscribeResponse">
2459         <wsdlsoap:body
2460             use="literal"/>
2461     </wsdl:output>
2462     <wsdl:fault name="NoSuchNameExceptionFault">
2463         <wsdlsoap:fault
2464             name="NoSuchNameExceptionFault"
2465             use="literal"/>
2466     </wsdl:fault>
2467     <wsdl:fault name="NoSuchSubscriberExceptionFault">
2468         <wsdlsoap:fault
2469             name="NoSuchSubscriberExceptionFault"
2470             use="literal"/>
2471     </wsdl:fault>
2472     <wsdl:fault name="InvalidURIExceptionFault">
2473         <wsdlsoap:fault
2474             name="InvalidURIExceptionFault"
2475             use="literal"/>
2476     </wsdl:fault>
2477     <wsdl:fault name="SecurityExceptionFault">
2478         <wsdlsoap:fault
2479             name="SecurityExceptionFault"
2480             use="literal"/>
2481     </wsdl:fault>
2482     <wsdl:fault name="ImplementationExceptionFault">
2483         <wsdlsoap:fault
2484             name="ImplementationExceptionFault"
2485             use="literal"/>
2486     </wsdl:fault>
2487 </wsdl:operation>
2488
2489 <wsdl:operation name="poll">
2490     <wsdlsoap:operation soapAction=""/>
2491     <wsdl:input name="pollRequest">
2492         <wsdlsoap:body
2493             use="literal"/>
2494     </wsdl:input>
2495     <wsdl:output name="pollResponse">
2496         <wsdlsoap:body
2497             use="literal"/>
2498     </wsdl:output>
2499     <wsdl:fault name="NoSuchNameExceptionFault">
2500         <wsdlsoap:fault
2501             name="NoSuchNameExceptionFault"
2502             use="literal"/>
2503     </wsdl:fault>
2504     <wsdl:fault name="SecurityExceptionFault">

```

```

2505         <wsdlsoap:fault
2506             name="SecurityExceptionFault"
2507             use="literal"/>
2508     </wsdl:fault>
2509     <wsdl:fault name="ImplementationExceptionFault">
2510         <wsdlsoap:fault
2511             name="ImplementationExceptionFault"
2512             use="literal"/>
2513     </wsdl:fault>
2514 </wsdl:operation>
2515
2516 <wsdl:operation name="immediate">
2517     <wsdlsoap:operation soapAction=""/>
2518     <wsdl:input name="immediateRequest">
2519         <wsdlsoap:body
2520             use="literal"/>
2521     </wsdl:input>
2522     <wsdl:output name="immediateResponse">
2523         <wsdlsoap:body
2524             use="literal"/>
2525     </wsdl:output>
2526     <wsdl:fault name="ECSpecValidationExceptionFault">
2527         <wsdlsoap:fault
2528             name="ECSpecValidationExceptionFault"
2529             use="literal"/>
2530     </wsdl:fault>
2531     <wsdl:fault name="SecurityExceptionFault">
2532         <wsdlsoap:fault
2533             name="SecurityExceptionFault"
2534             use="literal"/>
2535     </wsdl:fault>
2536     <wsdl:fault name="ImplementationExceptionFault">
2537         <wsdlsoap:fault
2538             name="ImplementationExceptionFault"
2539             use="literal"/>
2540     </wsdl:fault>
2541 </wsdl:operation>
2542
2543 <wsdl:operation name="getSubscribers">
2544     <wsdlsoap:operation soapAction=""/>
2545     <wsdl:input name="getSubscribersRequest">
2546         <wsdlsoap:body
2547             use="literal"/>
2548     </wsdl:input>
2549     <wsdl:output name="getSubscribersResponse">
2550         <wsdlsoap:body
2551             use="literal"/>
2552     </wsdl:output>
2553     <wsdl:fault name="NoSuchNameExceptionFault">
2554         <wsdlsoap:fault
2555             name="NoSuchNameExceptionFault"
2556             use="literal"/>
2557     </wsdl:fault>
2558     <wsdl:fault name="SecurityExceptionFault">
2559         <wsdlsoap:fault
2560             name="SecurityExceptionFault"
2561             use="literal"/>
2562     </wsdl:fault>
2563     <wsdl:fault name="ImplementationExceptionFault">
2564         <wsdlsoap:fault
2565             name="ImplementationExceptionFault"
2566             use="literal"/>
2567     </wsdl:fault>
2568 </wsdl:operation>
2569
2570 <wsdl:operation name="getStandardVersion">
2571     <wsdlsoap:operation soapAction=""/>
2572     <wsdl:input name="getStandardVersionRequest">
2573         <wsdlsoap:body
2574             use="literal"/>

```

```

2575         </wsdl:input>
2576         <wsdl:output name="getStandardVersionResponse">
2577             <wsdlsoap:body
2578                 use="literal"/>
2579         </wsdl:output>
2580         <wsdl:fault name="ImplementationExceptionFault">
2581             <wsdlsoap:fault
2582                 name="ImplementationExceptionFault"
2583                 use="literal"/>
2584         </wsdl:fault>
2585     </wsdl:operation>
2586
2587     <wsdl:operation name="getVendorVersion">
2588         <wsdlsoap:operation soapAction=""/>
2589         <wsdl:input name="getVendorVersionRequest">
2590             <wsdlsoap:body
2591                 use="literal"/>
2592         </wsdl:input>
2593         <wsdl:output name="getVendorVersionResponse">
2594             <wsdlsoap:body
2595                 use="literal"/>
2596         </wsdl:output>
2597         <wsdl:fault name="ImplementationExceptionFault">
2598             <wsdlsoap:fault
2599                 name="ImplementationExceptionFault"
2600                 use="literal"/>
2601         </wsdl:fault>
2602     </wsdl:operation>
2603 </wsdl:binding>
2604
2605 <!-- ALESERVICE -->
2606 <wsdl:service name="ALEService">
2607     <wsdl:port binding="impl:ALEServiceBinding" name="ALEServicePort">
2608         <!-- The value of the location attribute below is an example only;
2609             Implementations are free to choose any appropriate URL. -->
2610         <wsdlsoap:address
2611             location="http://localhost:6060/axis/services/ALEService"/>
2612     </wsdl:port>
2613 </wsdl:service>
2614
2615 </wsdl:definitions>

```

12 Use Cases (non-normative)

This section provides a non-normative illustration of how the ALE interface is used in various application scenarios.

1. For **shipment and receipt verification**, applications will request the number of Logistic Units such as Pallets and Cases moving through a portal, totaled by Pallet and Case GTIN across all serial numbers. Object types other than Pallet and Case should be filtered out of the reading.
2. For **retail OOS management**, applications will request one of 2 things:
 - a. The number of Items that were added to or removed from the shelf since the last read cycle, totaled by Item GTIN across all serial numbers. Object types other than Item should be filtered out of the reading; or
 - b. The total number of Items on the shelf during the current read cycle, totaled by GTIN across all serial numbers. Object types other than Item should be filtered out of the reading.
3. For **retail checkout**, applications will request the full EPC of Items that move through the checkout zone. Object types other than Item should be filtered out. In order to prevent charging for Items that aren't for sale (*e.g.*, Items the consumer or

checkout clerk brought into the store that inadvertently happen to be read), something in the architecture needs to make sure such Items are not read or filter them out. Prevention might be achievable with proper portal design and the ability for the checkout clerk to override errant reads. Alternatively, the ALE implementation could filter errant reads via access to a list of Items (down to the serial number) that are qualified for sale in that store (this could be hundreds of thousands to millions of items), or the POS application itself could do it. With the list options, the requesting application would be responsible for maintaining the list.

4. For **retail front door theft detection**, applications will request the full EPC of any Item that passes through the security point portal and that has not be marked as sold by the store and perhaps that meet certain theft detection criteria established by the store, such as item value. Like the retail checkout use case, the assumption is that the ALE implementation will have access to a list of store Items (to the serial number level) that have not been sold and that meet the stores theft alert conditions. The requesting application will be responsible for maintaining the list, and will decide what action, if any, should be taken based on variables such as the value and quantity of Items reported.
5. For **retail shelf theft detection**, applications will request the number of Items that were removed from the shelf since the last read cycle, totaled by Item GTIN across all serial numbers. Object types other than Item should be filtered out.
6. For **warehouse management**, a relatively complex range of operations and thus requirements will exist. For illustration at this stage, one of the requirements is that the application will request the EPC of the slot location into which a forklift operator has placed a Pallet of products. Object types other than “slot” should be filtered out of the reading.

The table below summarizes the ALE API settings used in each of these use cases.

Use Case	Event Cycle Boundaries	Report Settings		
		Result Set R	Filter $F(R)$	Report Type
1 (ship/rcpt)	Triggered by pallet entering and leaving portal	Complete	Pallet & Case	Group cardinality, $G = \text{pallet/case GTIN}$
2a (retail OOS)	Periodic	Additions & Deletions	Item	Group cardinality, $G = \text{item GTIN}$
2b (retail OOS)	Periodic	Complete	Item	Group cardinality, $G = \text{item GTIN}$
3 (retail ckout)	Single	Complete	Item	Membership (EPC)

Use Case	Event Cycle Boundaries	Report Settings		
		Result Set R	Filter $F(R)$	Report Type
4 (door theft)	Triggered by object(s) entering and leaving portal	Complete	None	Membership (EPC)
5 (shelf theft)	Periodic	Deletions	Item	Group cardinality, $G = \text{item GTIN}$
6 (forklift)	Single	Complete	Slot	Membership (EPC)

2660

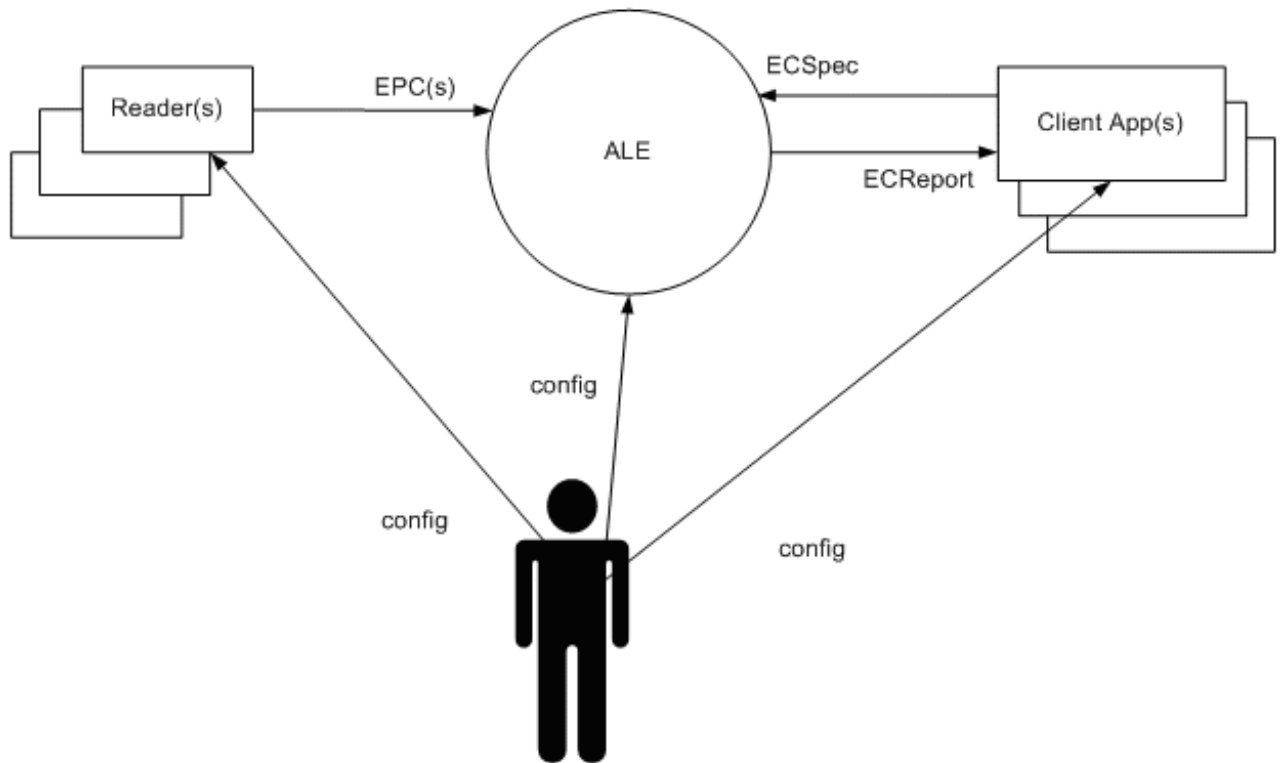
2661 13 ALE Scenarios (non-normative)

2662 This section provides a non-normative illustration of the API-level interactions between
2663 the ALE interface and the ALE client and other actors.

2664 13.1 ALE Context

2665 The ALE layer exists in a context including RFID readers, Users (administrative) and
2666 Client applications as shown below. Initially the administrators are responsible for
2667 installing and configuring the RFID environment. Once the environment is configured,
2668 EPC data (tag reads) are sent from the Readers to the ALE layer. In some cases the ALE
2669 layer may be implemented on the Reader or elsewhere, but in these scenarios we assume
2670 that the ALE layer is implemented as a distinct software component and is configured to
2671 support more than one Reader.

2672



2673

2674 The ALE clients are applications or services that process EPC tag information. Several
 2675 methods are defined within the ALE interface which allow clients to specify the data they
 2676 wish to receive and the conditions for the production of the reports containing the data.
 2677 These methods are:

- 2678 • define, undefine
- 2679 • subscribe, unsubscribe
- 2680 • poll
- 2681 • immediate
- 2682 • getECSpecNames, getECSpec

2683 These methods are defined normatively in Section 8.1.

2684 13.2 Scenarios

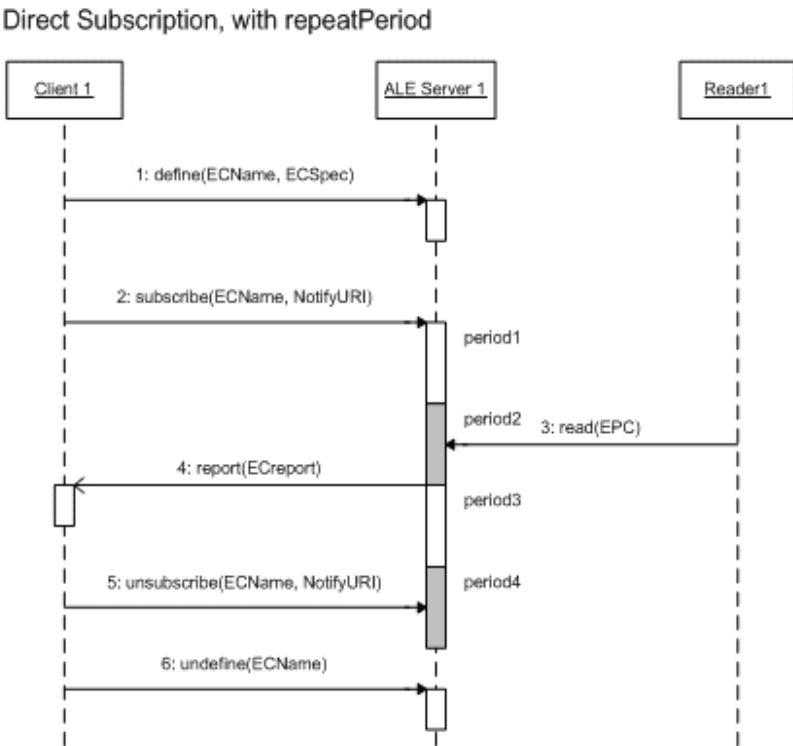
2685 A few sample scenarios are illustrated below to demonstrate the use of the ALE interface
 2686 messages. Below is a representative list of the kinds of scenarios ALE supports.

- 2687 1. Defining Subscribe ECName, ECSpec
 - 2688 a. Direct Subscription. Defined by A, Report to: A
 - 2689 b. Indirect Subscription Defined by A, Report to: B
- 2690 2. Poll(ECName)

- 2691 3. Immediate(ECSpec)
2692 4. Operation Errors
2693 5. System Errors

2694 **13.2.1 Scenario 1a: Direct Subscription**

2695 The scenario shown below involves a client application specifying the EPC data it is
2696 interested in collecting. After specifying the ECSpec, it then subscribes to receive the
2697 resulting EReports. The ECSpec shown in this scenario specifies that event cycles
2698 should repeat periodically. The EReportSpec requests reports for additions and
2699 deletions, and reportIfEmpty is set to false. This is a normal scenario involving no
2700 errors.



2701

2702 **13.2.1.1 Assumptions**

- 2703 1. All discovery, configuration, and initialization required has already been
2704 performed.
2705 2. The ALE layer is implemented as a distinct software component.
2706 3. ECSpec boundary condition specified using: repeatPeriod
2707 4. ECFilterSpec includePatterns includes the EPC(s) illustrated in
2708 this scenario
2709 5. Client 1 is the only client of ALE and the only subscriber of the ECSpec

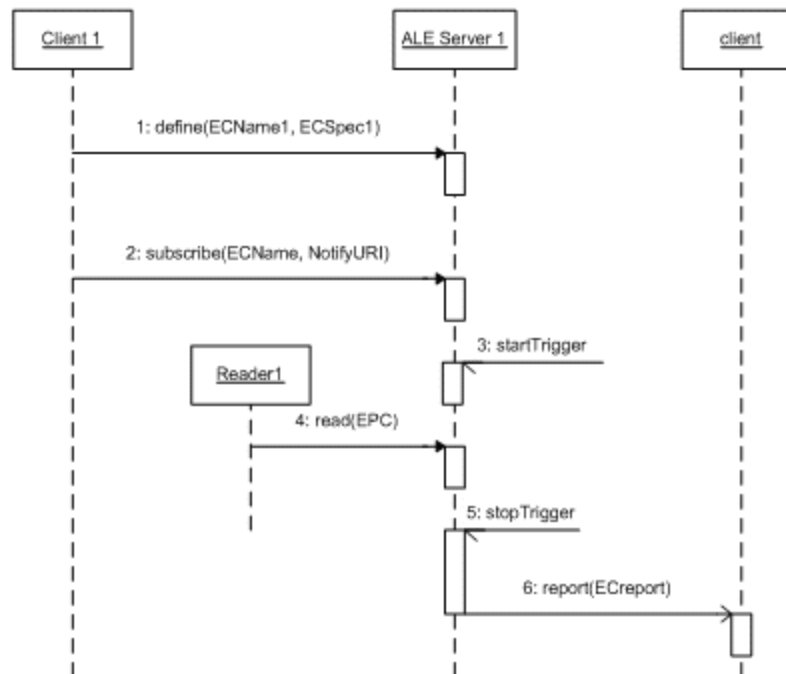
2710 13.2.1.2 Description

- 2711 1. The client calls the `define` method of the ALE interface. The `ECSpec`
2712 specifies that the event cycle is to begin using `repeatPeriod` as the
2713 boundary specification and to end using `duration` as the boundary
2714 specification (but any valid boundary conditions could be specified). The
2715 `ECReportSpec` and `ECFilterSpec` contained within the `ECSpec` are
2716 defined to include the EPC data sent later in step 3.
- 2717 2. The client calls the `subscribe` method of the ALE interface, including a
2718 URI that identifies the client itself as the destination for the `ECReports`. At
2719 this point the `ECSpec` is considered “Requested.” Since the start condition is
2720 given by `repeatPeriod`, the `ECSpec` immediately transitions to the
2721 “Active” state.
- 2722 3. During `period1` no new tags (additions) were reported by the Reader, and no
2723 deletions were noted, thus no `ECReports` is generated.
- 2724 4. In `period2`, an EPC that does meet the filter conditions specified in the
2725 `ECSpec` is reported to the ALE layer by one of the Readers indicated in the
2726 `ECSpec`.
- 2727 5. At the end of `period2`, the requested `ECReports` is generated and sent to the
2728 client.
- 2729 6. In `period3`, no EPCs are reported, and no `ECReports` are generated.
- 2730 7. In `period4` the client calls the `unsubscribe` method of the ALE interface.
2731 As this client is the only subscriber, the `ECSpec` transitions to the
2732 “Unrequested” state, and no further `ECReports` are generated.
- 2733 8. Because the `ECSpec` is Unrequested, the client can undefine the `ECSpec`
2734 without any error.

2735 13.2.2 Scenario 1b: Indirect Subscription

2736 The scenario shown below involves a client application specifying the EPC data that is of
2737 interest to another observer. After specifying the `ECSpec`, the client subscribes a third
2738 party observer to receive the resulting `ECReports`. The `ECSpec` shown in this
2739 scenario specifies the event cycle to start and stop using a trigger mechanism. This is a
2740 normal scenario involving no errors.

Indirect Subscription, with Triggers



2741

2742 13.2.2.1 Assumptions

- 2743 1. All discovery, configuration, and initialization required has already been
- 2744 performed.
- 2745 2. The ALE layer is implemented as a distinct software component.
- 2746 3. ECSpec boundary conditions specified using startTrigger, stopTrigger
- 2747 4. ECFilterSpec includePatterns includes the EPC(s) illustrated in
- 2748 this scenario

2749 13.2.2.2 Description

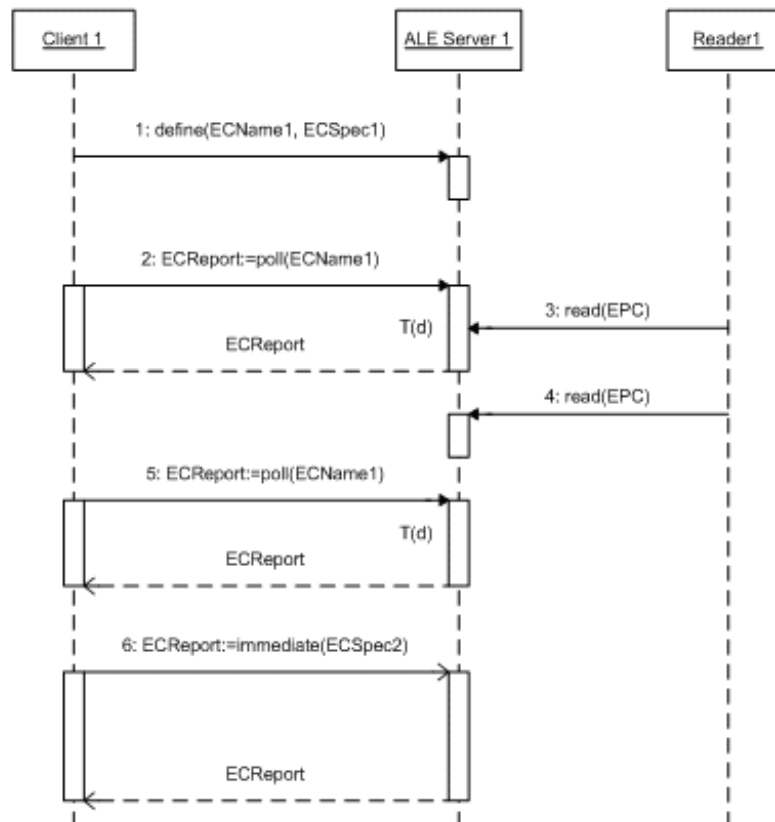
- 2750 1. The ALE client calls the define methods of the ALE interface. The
- 2751 ECSpec contains a valid startTrigger and stopTrigger as boundary
- 2752 specifications – though any valid boundary conditions could be specified. The
- 2753 ECRreportSpec and ECFilterSpec contained within the ECSpec is
- 2754 defined to include the EPC data sent later in step 4.
- 2755 2. The ALE client calls the subscribe method of the ALE interface which
- 2756 includes the URI of the intended observer. At this point the ECSpec is
- 2757 considered “Requested.”
- 2758 3. After the start trigger is received, the ECSpec is considered “Active.”
- 2759 Subsequent EPCs that meet the filter conditions in the ECSpec will be
- 2760 collected by the ALE layer.
- 2761 4. An EPC that does meet the filter conditions in the ECSpec is reported to the
- 2762 ALE layer.

- 2763 5. The stop trigger is received. The ECSpec transitions to the “Requested”
 2764 state.
 2765 6. The ECReports is generated and sent asynchronously to the observer.

2766 13.2.3 Scenario 2, 3: Poll, Immediate

2767 The scenario shown illustrates an ALE client using the poll method of the ALE
 2768 interface to synchronously obtain the EPC data it is interested in collecting. The
 2769 ECSpec shown in this scenario specifies the event cycle boundary to be a duration of
 2770 time. Later in the scenario the ALE client uses the immediate method of the ALE
 2771 interface, again synchronously obtaining EPC data. The combination of poll and
 2772 immediate is not required, and only serves to illustrate a possibility. This is a normal
 2773 scenario involving no errors.

Poll, Immediate, with duration



2774

2775 13.2.3.1 Assumptions

- 2776 1. All discovery, configuration, and initialization required has already been
 2777 performed.
 2778 2. The ALE layer is implemented as a distinct software component.
 2779 3. ECSpec boundary condition is specified as duration.

2780 4. `ECFilterSpec includePatterns` includes the EPC(s) illustrated in
2781 this scenario.

2782 13.2.3.2 Description

- 2783 1. The ALE client calls the `define` method of the ALE interface. The
2784 `ECSpec` contains a valid `duration` as the boundary specification – though
2785 any valid boundary conditions could be specified. The `ECReportSpec` and
2786 `ECFilterSpec` contained within the `ECSpec` are defined to include the
2787 EPC data sent later in steps 3 and 4. At this point the `ECSpec` is considered
2788 “Unrequested.”
- 2789 2. The ALE client calls the `poll` method of the ALE interface, naming the
2790 `ECSpec` previously defined in Step 1. At this point the `ECSpec` is
2791 transitioned to the “Active” state, and the event cycle begins for the duration
2792 specified in the `ECSpec`. During the duration of the event cycle the ALE
2793 client is blocked waiting for a response to the `poll` method.
- 2794 3. An EPC which meets the filter conditions of the `ECSpec` is received during
2795 the event cycle. At the end of the event cycle, the `ECReports` is generated
2796 and returned to the ALE client as the response to the `poll` method. At this
2797 point the `ECSpec` transitions to the “Unrequested” state.
- 2798 4. An EPC that meets the filter conditions of the `ECSpec` is reported to the ALE
2799 layer, but since there is no “Active” `ECSpec`, this EPC will not be reported.
- 2800 5. The ALE client invokes the `poll` method of the ALE interface a second time.
2801 This is similar to the process described above in Steps 2 and 3, but since no
2802 EPC is received, no EPC data is returned in the `ECReports`.
- 2803 6. Later, the ALE client calls the `immediate` method of the ALE interface.
2804 This is very similar to the use of `poll`, except that when the client calls
2805 `immediate` it provides the `ECSpec` as part of the method call, as opposed
2806 to referring to a previously defined `ECSpec`. Since a new `ECSpec` is
2807 provided with the `immediate` method, it can contain any valid combination
2808 of parameters and report options.
2809

2810 14 Glossary (non-normative)

2811 This section provides a non-normative summary of terms used within this specification.
2812 For normative definitions of these terms, please consult the relevant sections of the
2813 document.

Term	Section	Meaning
ALE (Application Level Events) Interface	1	Software interface through which ALE Clients may obtain filtered, consolidated EPC data from a variety of sources.

Term	Section	Meaning
ALE (Application Level Events) Layer	2	Functionality that sits between raw EPC detection events (RFID tag reads or otherwise) and application business logic (an ALE Client). The ALE Interface is the interface between this layer and the ALE Client.
ALE Client	2	Software, typically application business logic, which obtains EPC data through the ALE Interface.
Event Cycle	3	One or more Read Cycles, from one or more Readers, that are to be treated as a unit from a client perspective. It is the smallest unit of interaction between the ALE Interface and an ALE Client.
Read Cycle	3	The smallest unit of interaction of the ALE Layer with a Reader.
Reader	3	A source of raw EPC data events. Often an RFID reader, but may also be EPC-compatible bar code reader, or even a person typing on a keyboard.
Report	3	Data about event cycle communicated from the ALE interface to an ALE Client.
Immediate Request	2	A request in which information is reported on a one-time basis at the time of request. Immediate requests are made using the <code>immediate</code> or <code>poll</code> methods of the ALE Interface.
Recurring Request	2	A request in which information is reported repeatedly whenever an event is detected or at a specified time interval. Recurring requests are made using the <code>subscribe</code> method of the ALE Interface.
Grouping Operator	5	A function that maps an EPC code into a group code. Specifies how EPCs read within an Event Cycle are to be partitioned into groups for reporting purposes.
Physical Reader	7	A physical device, such as an RFID reader or bar code scanner, that acts as one or more Readers for the purposes of the ALE Layer.
Logical Reader Name	7	An abstract name that an ALE Client uses to refer to one or more Readers that have a single logical purpose; <i>e.g.</i> , <code>DockDoor42</code> .

2814

2815 15 References

2816 [ISODir2] ISO, "Rules for the structure and drafting of International Standards
2817 (ISO/IEC Directives, Part 2, 2001, 4th edition)," July 2002.

2818 [RFC1738] T. Berners-Lee, L. Masinter, M. McCahill, "Uniform Resource Locators
2819 (URL)," RFC 1738, December 1994, <http://www.ietf.org/rfc/rfc1738>.

2820 [RFC2396] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers
2821 (URI): Generic Syntax," RFC2396, August 1998, <http://www.ietf.org/rfc/rfc2396>.

2822 [RFC2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T.
2823 Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," RFC2616, June 1999,
2824 <http://www.ietf.org/rfc/rfc2616>.

2825 [Savant0.1] Oat Systems and MIT Auto-ID Center, "The Savant Version 0.1 (Alpha),"
2826 MIT Auto-ID Center Technical Manual MIT-AUTO-AUTOID-TM-003, February 2002,
2827 <http://www.autoidlabs.org/whitepapers/MIT-AUTOID-TM-003.pdf>.

2828 [Savant1.0] S. Clark, K. Traub, D. Anarkat, T. Osinski, E. Shek, S. Ramachandran, R.
2829 Kerth, J. Weng, B. Tracey, "Auto-ID Savant Specification 1.0," Auto-ID Center Software
2830 Action Group Working Draft WD-savant-1_0-20031014, October 2003.

2831 [TDS1.1] EPCglobal, "EPC Tag Data Standards Version 1.1 Rev.1.24," EPCglobal
2832 Standard Specification, April 2004,
2833 [http://www.epcglobalinc.org/standards_technology/EPCTagDataSpecification11rev124.p](http://www.epcglobalinc.org/standards_technology/EPCTagDataSpecification11rev124.pdf)
2834 [df](http://www.epcglobalinc.org/standards_technology/EPCTagDataSpecification11rev124.pdf).

2835 [WSDL1.1] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, "Web Services
2836 Description Language (WSDL) 1.1," W3C Note, March 2001,
2837 <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.

2838 [WSI] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, P. Yendluri, "Basic
2839 Profile Version 1.0," WS-i Final Material, April 2004, [http://www.ws-](http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html)
2840 [i.org/Profiles/BasicProfile-1.0-2004-04-16.html](http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html).

2841 [XML1.0] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau,
2842 "Extensible Markup Language (XML) 1.0 (Third Edition)," W3C Recommendation,
2843 February 2004, <http://www.w3.org/TR/2004/REC-xml-20040204/>.

2844 [XSD1] H. Thompson, D. Beech, M. Maloney, N. Mendelsohn, "XML Schema Part 1:
2845 Structures," W3C Recommendation, May 2001, <http://www.w3.org/TR/xmlschema-1/>.

2846 [XSD2] P. Biron, A. Malhotra, "XML Schema Part 2: Datatypes," W3C
2847 Recommendation, May 2001, <http://www.w3.org/TR/xmlschema-2/>.

2848 [XMLVersioning] D. Orchard, "Versioning XML Vocabularies," December 2003,
2849 <http://www.xml.com/pub/a/2003/12/03/versioning.html>.

2850