# 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.

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

# Audience for this document

The target audience for this specification includes:

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

# Status of this document

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

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

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

# Table of Contents

# 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 Massachussetts 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:

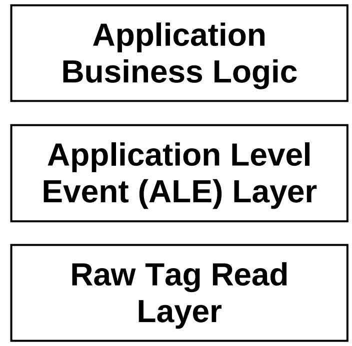
1. Drive as much filtering and counting of reads as low in the architecture as possible (*i.e.*, in first preference to readers, then to "middleware", and as a last resort to "applications"), while meeting application and cost needs;

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).




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.

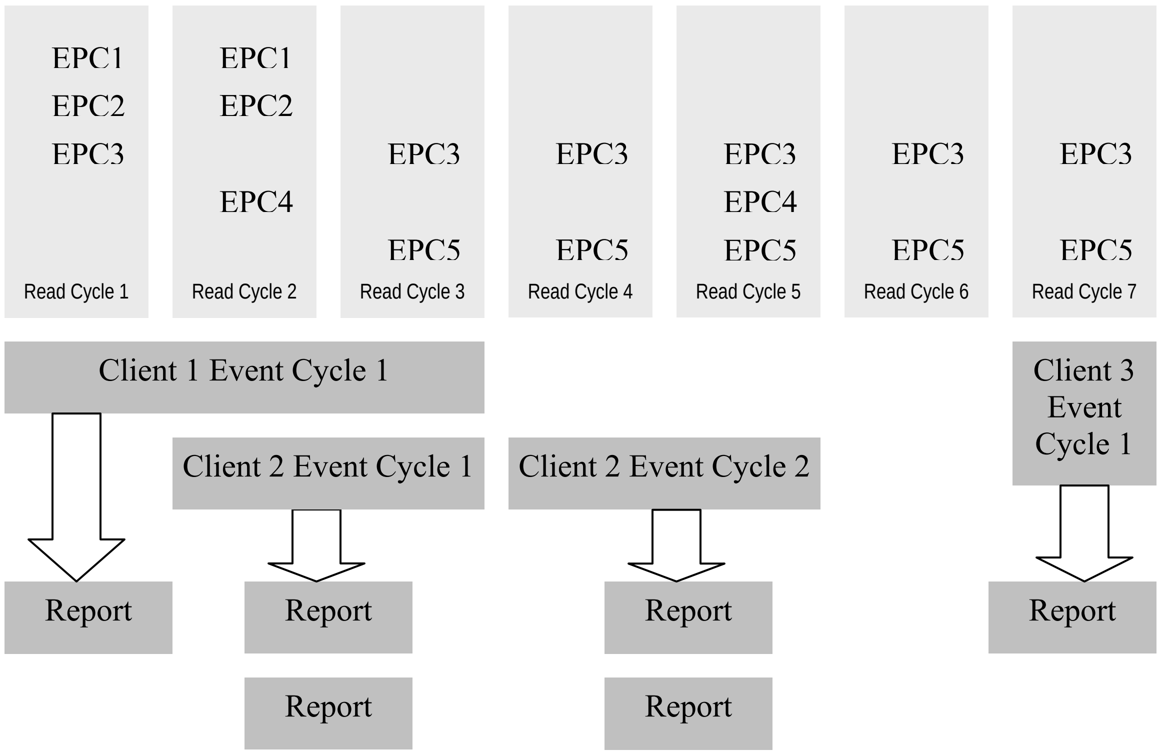
A *report* is data about an event cycle communicated from the ALE implementation to a client. The report is the output of the ALE layer, communicated to the Application Business Logic Layer.

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:




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

The set of EPCs in a given read cycle from a given Reader is denoted by *S*. In the picture above, $S1$ = {EPC1, EPC2, EPC3} and $S2$ = {EPC1, EPC2, EPC4}.

An event cycle is treated as a unit by clients, so clients do not see any of the internal structure of the event cycle. All that is relevant, therefore, is the complete set of EPCs occurring in any of the read cycles that make up the event cycle, from any of the Readers in the set specified for the event cycle, with duplicates removed. This is simply the union of the read cycle sets: $E = S1 \cup S2 \cup \ldots$. In the example above for Client 1 Event Cycle 1 we have $E1.1$ = {EPC1, EPC2, EPC3, EPC4, EPC5}.

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

- What set *R* to report, which may be
  - The *complete* set from the current event cycle $R = Ecur$; or

    - The *differential* set that only includes differences of the current event cycle relative to the previous one (assuming the same event cycle boundaries). This can be the set of additions $R = (Ecur - Eprev)$ or the set of deletions $R = (Eprev - Ecur)$, where '–' denotes the set difference operator.

- An optional filter *F(R)* to apply, including as part of the standard ALE interface:
    - One or more object types derived from the "filter bits" of the EPC Tag Data Standard [TDS1.1], including "product" objects (*e.g.*, pallet, case, *etc.*) as well as "location" objects (*e.g.*, warehouse slots, trucks, retail shelves, *etc.*, that contain embedded EPC tags)
    - A specific list of EPCs
    - A range of EPCs
- Whether to report
    - The members of the set, *F(R)* (*i.e.*, the EPCs themselves), possibly grouped as described in Section 5, and in what format (e.g., pure identity URI, tag URI, raw binary, etc);
    - The quantity, or cardinality, of the set |*F(R)*|, or of the groups making up the set as described in Section 5.

The available options are described normatively in Section 8.2.

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

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

Note that because the filtering operations commute with the set union and difference operations, there is a great deal of freedom in how an ALE implementation actually carries out the task of fulfilling a report request. For example, in one implementation, there may be a Reader that is capable of doing filtering directly within the Reader, while in a second implementation the Reader may not be capable of filtering and so software implementing the ALE API must do it. But the ALE API itself need not change – the client specifies the reports, and the implementation of the API decides where best to carry out the requested filtering.

# 5 Group Reports

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

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

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

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

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

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

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

# 6 Read Cycle Timing

The ALE API is intentionally silent about the timing of read cycles. Clients may specify the boundaries of event cycles, which accumulate data from one or more underlying read cycles, but the API does not provide a client with explicit control over the frequency at 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** |
| `DockDoor42` | Acme42926 | 0 | UHF |
| | Acme42926 | 1 | UHF |
| | Acme43629 | 0 | UHF |
| `DockDoor43` | Acme44926 | 0 | UHF |
| | Acme44926 | 1 | UHF |
| | Acme49256 | 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 notificationURI. Different ALE implementations MAY provide a variety of available notification means (*e.g.*, JMS, MQ-Series, TIBCO, e-mail, SOAP, *etc.*); this is intended to be a point of extensibility. Section 9 specifies notification means that are standardized, and specifies the conformance requirement (MAY, SHOULD, SHALL) for each.

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

```
dataMember1 : Type1
dataMember2 : Type2
---
method1(ArgName:ArgType, ArgName:ArgType, …) : ReturnType
method2(ArgName:ArgType, ArgName:ArgType, …) : ReturnType
```

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

In the case of the standard XML bindings for ECSpec and ECReports, the extension points are implemented within the XML schema following the methodology described in Section 10.1. In the case of the standard SOAP binding for the ALE interface, the 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:

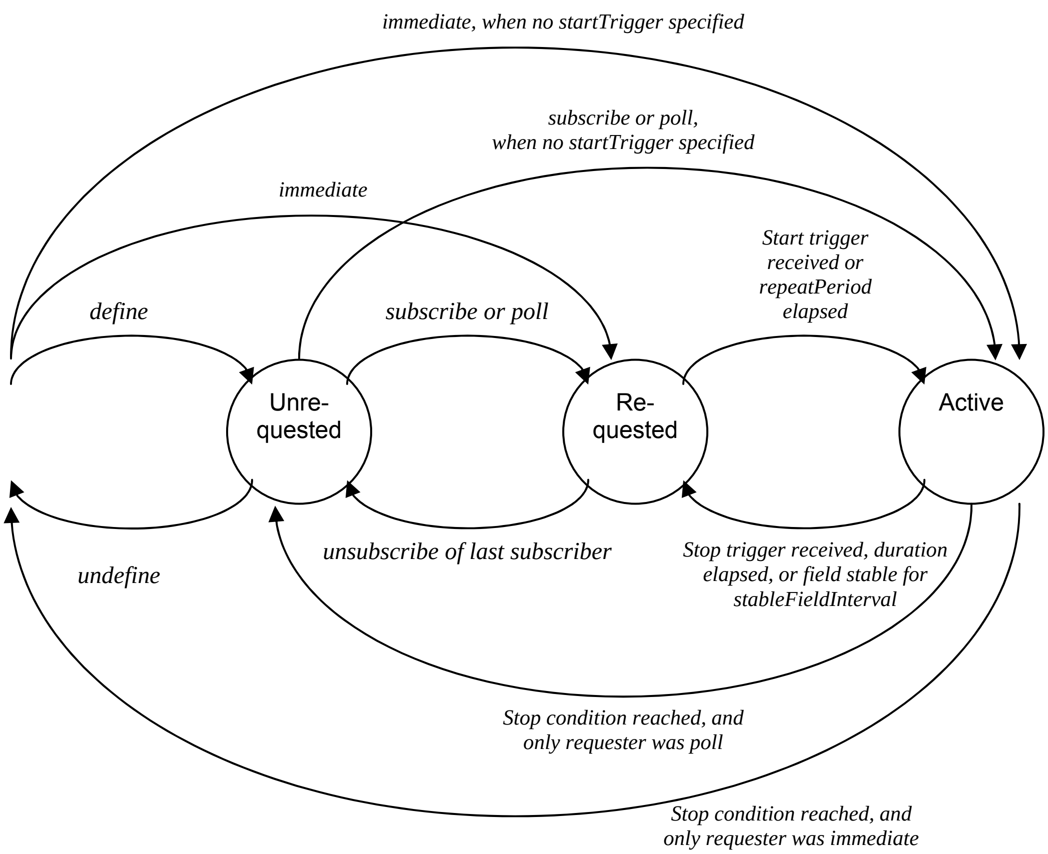
- It has previously been defined using `define`, it has not yet been `undefine`d, and there has been at least one `subscribe` call for which there has not yet been a corresponding `unsubscribe` call.

- It has previously been defined using `define`, it has not yet been `undefine`d, 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. `ECSpec`s 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 `subscribe`d 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 `ECReports`, which contains one or more reports generated from one activation of an `ECSpec`. `ECReports` 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 `ECReports` 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 return a string corresponding to a version of this specification to which the implementation fully complies, and SHOULD return the string corresponding to the latest version to which it complies. To indicate compliance with this Version 1.0 of the ALE specification, the implementation SHALL return the string `1.0`. The method `getVendorVersion` returns a string that identifies what vendor extensions this implementation provides. The possible values of this string and their meanings are vendor-defined, except that the empty string SHALL indicate that the implementation implements only standard functionality with no vendor extensions. When an implementation chooses to return a non-empty string, the value returned SHALL be a URI where the vendor is the owning authority. For example, this may be an HTTP URL whose authority portion is a domain name owned by the vendor, a URN having a URN namespace identifier issued to the vendor by IANA, an OID URN whose initial path is a Private Enterprise Number assigned to the vendor, etc.

### 8.1.1 Error Conditions

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

| Exception Name | Meaning |
|---|---|
| `SecurityException` | 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. |
| `DuplicateNameException` | The specified ECSpec name already exists. |
| `ECSpecValidationException` | The specified ECSpec is invalid; *e.g.*, 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. |
| `InvalidURIException` | 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. |
| `NoSuchNameException` | The specified ECSpec name does not exist. |
| `NoSuchSubscriberException` | 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. |

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

| ALE Method | Exceptions |
|---|---|
| define | DuplicateNameException<br>ECSpecValidationException<br>SecurityException<br>ImplementationException |
| undefine | NoSuchNameException<br>SecurityException<br>ImplementationException |
| getECSpec | NoSuchNameException<br>SecurityException<br>ImplementationException |
| getECSpecNames | SecurityException<br>ImplementationException |
| subscribe | NoSuchNameException<br>InvalidURIException<br>DuplicateSubscriptionException<br>SecurityException<br>ImplementationException |
| unsubscribe | NoSuchNameException<br>NoSuchSubscriberException<br>InvalidURIException<br>SecurityException<br>ImplementationException |

| ALE Method | Exceptions |
|---|---|
| `poll` | `NoSuchNameException`<br>`SecurityException`<br>`ImplementationException` |
| `immediate` | `ECSpecValidationException`<br>`SecurityException`<br>`ImplementationException` |
| `getSubscribers` | `NoSuchNameException`<br>`SecurityException`<br>`ImplementationException` |

## 8.2 ECSpec

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

```
readers : List   // List of logical reader names
boundaries : ECBoundarySpec
reportSpecs : List    // List of one or more ECReportSpec
                      //  instances
includeSpecInReports : boolean
<<extension point>>
---
```

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

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

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

If an `ECSpec` has `includeSpecInReports` set to true, then the ALE implementation SHALL include the complete `ECSpec` as part of every `ECReports` 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 `ECSpecValidationException`. 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 `ECSpecValidationException`.

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 ECSpec 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 ECSpec 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 ECSpec has never transitioned to the *unrequested* state.
- If neither `startTrigger` nor `repeatPeriod` are specified, an event cycle is started when:
    - The ECSpec transitions from the *unrequested* state to the *requested* state; or
    - Immediately after the previous event cycle, if the ECSpec is in the *requested* state.

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

- The duration, when specified, expires.
- When the stableSetInterval is specified, no *new* EPCs are reported by any Reader for the specified interval (*i.e.*, the set of EPCs being accumulated by the event cycle is stable for the specified interval). In this context, "new" is to be interpreted collectively among Readers contributing to this event cycle. For example, suppose a given event cycle is accumulating data from Readers A and B. If Reader A completes a read cycle containing EPC X, then subsequently Reader B completes a different read cycle containing the same EPC X, then the occurrence of EPC X in B's read cycle is not considered "new" for the purposes of evaluating the stableSetInterval. Note that in the context of the stableSetInterval, the term "stable" only implies that no *new* EPCs are detected; it does not imply that previously detected EPCs must continue to be detected. That is, only *additions*, and not *deletions*, are considered in determining that the EPC set is "stable."
- The stopTrigger, when specified, is received.
- The ECSpec transitions to the *unrequested* state.

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

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

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

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

The ECTrigger values startTrigger and stopTrigger, if specified, must conform to URI syntax as defined by [RFC2396], and must be supported by the ALE implementation; otherwise, the define and immediate methods SHALL raise an 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 the EPC URI pattern syntax specified in [TDS1.1], then the define and immediate methods SHALL raise an ECSpecValidationException.

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

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

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

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

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

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

### 8.2.6 ECReportSetSpec

`ECReportSetSpec` 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:

$$
\begin{aligned}
F(R) = \{\ epc \mid\ & epc \in R \\
& \&\ (epc \in I_1 \mid \ldots \mid epc\ \mathrm{i}{\in}\ I_n) \\
& \&\ epc \notin E_1\ \&\ \ldots\ \&\ epc \notin E_n\ \}
\end{aligned}
$$

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.

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

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

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

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

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

### 8.2.9 ECGroupSpec

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

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

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

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

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

But the following are not:

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

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. |
| *`Number`* | Only EPCs having *Number* in this field will belong to this group. |
| `[`*`Lo-Hi`*`]` | Only EPCs whose value for this field falls within the specified range will belong to this group. |

Here are examples of pattern URIs used as group operators:

| **Pattern URI** | **Meaning** |
|---|---|
| `urn:epc:pat:sgtin-64:X.*.*.*` | groups by filter value (*e.g.*, 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.e., 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 |

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

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

```
urn:epc:tag:sgtin-64:3.0036000.123456.400
urn:epc:tag:sgtin-64:3.0036000.123456.500
```

```
urn:epc:tag:sgtin-64:3.0029000.111111.100
urn:epc:tag:sscc-64:3.0012345.31415926
```

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

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

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`<br>`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` |

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

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

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

Pat_i = `urn:epc:pat:`*`type_i`*`:`*`field_i_1`*`.`*`field_i_2`*`.`*`field_i_3`*`...`
Pat_j = `urn:epc:pat:`*`type_j`*`:`*`field_j_1`*`.`*`field_j_2`*`.`*`field_j_3`*`...`

Then Pat_i and Pat_j are disjoint if:

- *`type_i`* is not equal to *`type_j`*
- *`type_i`* = *`type_j`* but there is at least one field *k* for which *`field_i_k`* and *`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 |

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

G = (Pat_1, Pat_2, ..., Pat_N)

Let each pattern be written as a series of fields:

Pat_i = urn:epc:pat:*type_i*:*field_i_1*.*field_i_2*.*field_i_3*...

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

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

urn:epc:tag:*type_epc*:*field_epc_1*.*field_epc_2*.*field_epc_3*...

The epc is said to *match* Pat_i if

- *type_epc* = *type_i*; and
- For each field *k*, one of the following is true:
    - *field_i_k* = X
    - *field_i_k* = *
    - *field_i_k* is a number, equal to *field_epc_k*
    - *field_i_k* is a range [*Lo*-*Hi*], and *Lo* ≤ *field_epc_k* ≤ *Hi*

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

G(epc) is then defined as follows:

- If epc matches Pat_i for some i, then

  G(epc) = urn:epc:pat:*type_epc*:*field_g_1*.*field_g_2*.*field_g_3*...

  where for each *k*, *field_g_k* = *, if *field_i_k* = *; or *field_g_k* = *field_epc_j*, otherwise
- If epc does not match Pat_i for any i, then G(epc) = the default group.

### 8.2.10 ECReportOutputSpec

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

```
includeEPC : boolean
includeTag : boolean
includeRawHex : boolean
includeRawDecimal : boolean
includeCount : boolean
<<extension point>>
---
```

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

### 8.2.11 Validation of ECSpecs

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

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

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

## 8.3 ECReports

`ECReports` is the output from an event cycle.

```
specName : string
date : dateTime
ALEID : string
totalMilliseconds : long
terminationCondition : ECTerminationCondition
spec : ECSpec
reports : List   // List of ECReport
<<extension point>>
---
```

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

| Field | Description |
|---|---|
| `specName` | The name of the ECSpec that controlled this event cycle. In the case of an ECSpec that was requested using the `immediate` method (Section 8.1), this name is one chosen by the ALE implementation. |
| `date` | 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. |
| `ALEID` | 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 `ECReports` instance. Only included if the `ECSpec` has `includeSpecInReports` set to true. |

### 8.3.1 ECTerminationCondition

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

```
<<Enumerated Type>>
TRIGGER
DURATION
STABLE_SET
UNREQUEST
```

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

### 8.3.2 ECReport

`ECReport` represents a single report within an event cycle.

```
reportName : string
groups : List     // List of ECReportGroup instances
<<extension point>>
---
```

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

### 8.3.3 ECReportGroup

`ECReportGroup` represents one group within an `ECReport`.

```
groupName : string
groupList : ECReportGroupList
groupCount : ECReportGroupCount
<<extension point>>
---
```

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

### 8.3.4 ECReportGroupList

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

```
members : List //List of ECReportGroupListMember instances
<<extension point>>
---
```

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

### 8.3.5 ECReportGroupListMember

Each member of the `ECReportGroupList` is an `ECReportGroupListMember` as defined below. The reason for having `ECReportGroupListMember` 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 `ECReportOutputSpec`. Specifically, the `epc` field SHALL be non-null if and only if the `includeEPC` field of `ECReportOutputSpec` 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 ECReportGroupListMember element in the ECReportGroupList, even if those ECReportGroupListMember 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 ECReportOutputSpec specified includeEPC true and all other formats false, then the resulting ECReportGroupList SHALL have two ECReportGroupListMember 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 ECReportGroupCount

An ECReportGroupCount is included in an ECReportGroup when the includeCount field of the corresponding ECReportOutputSpec 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 ECReports in XML via the HTTP protocol using the POST operation. Implementations SHOULD provide support for this notification URI.

The syntax for HTTP notification URIs as used by ALE is defined in [RFC2616], Section 3.2.2. Informally, an HTTP URI has one of the two following forms:

```
http://host:port/remainder-of-URL
http://host/remainder-of-URL
```

where

- *host* is the DNS name or IP address of the host where the receiver is listening for incoming HTTP connections.
- *port* is the TCP port on which the receiver is listening for incoming HTTP connections. The port and the preceding colon character may be omitted, in which case the port defaults to 80.
- *remainder-of-URL* is the URL to which an HTTP POST operation will be directed.

The ALE implementation delivers event cycle reports by sending an HTTP POST request to receiver designated in the URI, where *remainder-of-URL* is included in the HTTP `request-line` (as defined in [RFC2616]), and where the payload is the `ECReports` instance encoded in XML according to the schema specified in Section 10.2.

The interpretation by the ALE implementation of the response code returned by the receiver is outside the scope of this specification; however, all implementations SHALL interpret a response code 2xx (that is, any response code between 200 and 299, inclusive) as a normal response, not indicative of any error.

## 9.2 TCP Notification URI

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

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

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

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

Informally, a TCP URI has the following form:

```
tcp://host:port
```

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

## 9.3 FILE Notification URI

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

The syntax for FILE notification URIs as used by ALE is defined in [RFC1738], Section 3.10. Informally, an FILE URI has one of the two following forms:

```
file://host/path
file:///path
```

where

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

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

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

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

# 10 XML Schema for Event Cycle Specs and Reports

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

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

## 10.1 Extensibility Mechanism

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

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

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

- For each type in which <<extension point>> occurs, include an `xsd:anyAttribute` declaration. This declaration provides for the addition of new attributes, either in subsequent versions of the standard schema or in vendor-specific schema.

- For each type in which <<extension point>> occurs, include an optional (`minOccurs` = 0) element named `extension`. The type declared for the `extension` element will always be as follows:

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

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

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

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

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

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

- Vendor-specific attributes may be added to any type in which <<extension point>> occurs. Vendor-specific attributes SHALL NOT be in the EPCglobal ALE namespace (`urn:epcglobal:ale:xsd:1`). Vendor-specific attributes SHALL be in a namespace whose namespace URI has the vendor as the owning authority. (In schema parlance, this means that all vendor-specific attributes must have `qualified` as their form.) For example, the namespace URI may be an HTTP URL whose authority portion is a domain name owned by the vendor, a URN having a URN namespace identifier issued to the vendor by IANA, an OID URN whose initial path is a Private Enterprise Number assigned to the vendor, etc. Declarations of vendor-specific attributes SHALL specify `use="optional"`.

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

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

```
 <xsd:group name="VendorExtension">
   <xsd:sequence>
     <!--
       Definitions or references to vendor elements
       go here.  Each SHALL specify minOccurs="0".
     -->
     <xsd:any processContents="lax"
              minOccurs="0" maxOccurs="unbounded"
              namespace="##other"/>
   </xsd:sequence>
</xsd:group>
```

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

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

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

- Standard attributes may be added to any type in which `<<extension point>>` occurs. Standard attributes SHALL NOT be in any namespace, and SHALL NOT 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>
```

```
        <xsd:anyAttribute processContents="lax"/>
    </xsd:complexType>

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

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

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

    <xsd:complexType name="ECLogicalReaders">
        <xsd:sequence>
            <xsd:element name="logicalReader" type="xsd:string" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ECReport">
        <xsd:sequence>
           <xsd:element name="group" type="ale:ECReportGroup" minOccurs="0"
maxOccurs="unbounded"/>
          <xsd:element name="extension" type="ale:ECReportExtension"
                       minOccurs="0"/>
          <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                   namespace="##other"/>
        </xsd:sequence>
        <xsd:attribute name="reportName" type="xsd:string" use="required"/>
        <xsd:anyAttribute processContents="lax"/>
    </xsd:complexType>

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

    <xsd:complexType name="ECReportList">
       <xsd:sequence>
          <xsd:element name="report" type="ale:ECReport" minOccurs="0"
maxOccurs="unbounded"/>
       </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ECReportGroup">
       <xsd:sequence>
          <xsd:element name="groupList" type="ale:ECReportGroupList" minOccurs="0"/>
          <xsd:element name="groupCount" type="ale:ECReportGroupCount" minOccurs="0"/>
          <xsd:element name="extension" type="ale:ECReportGroupExtension"
                       minOccurs="0"/>
          <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                   namespace="##other"/>
       </xsd:sequence>
       <!-- The groupName attribute SHALL be omitted to indicate the default group. -->
       <xsd:attribute name="groupName" type="xsd:string" use="optional"/>
```

```
      <xsd:anyAttribute processContents="lax"/>
   </xsd:complexType>

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

   <xsd:complexType name="ECReportGroupList">
      <xsd:sequence>
         <xsd:element name="member" type="ale:ECReportGroupListMember"
                      minOccurs="0" maxOccurs="unbounded"/>
         <xsd:element name="extension" type="ale:ECReportGroupListExtension"
                      minOccurs="0"/>
         <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                  namespace="##other"/>
      </xsd:sequence>
   </xsd:complexType>

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

   <xsd:complexType name="ECReportGroupListMember">
      <xsd:sequence>
         <!-- Each of the following four elements SHALL be omitted if null. -->
         <xsd:element name="epc" type="epcglobal:EPC" minOccurs="0"/>
         <xsd:element name="tag" type="epcglobal:EPC" minOccurs="0"/>
         <xsd:element name="rawHex" type="epcglobal:EPC" minOccurs="0"/>
         <xsd:element name="rawDecimal" type="epcglobal:EPC" minOccurs="0"/>
         <xsd:element name="extension" type="ale:ECReportGroupListMemberExtension"
                      minOccurs="0"/>
         <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                  namespace="##other"/>
      </xsd:sequence>
      <xsd:anyAttribute processContents="lax"/>
   </xsd:complexType>

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

   <xsd:complexType name="ECReportGroupCount">
       <xsd:sequence>
           <xsd:element name="count" type="xsd:int"/>
           <xsd:element name="extension" type="ale:ECReportGroupCountExtension"
                        minOccurs="0"/>
           <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                    namespace="##other"/>
       </xsd:sequence>
       <xsd:anyAttribute processContents="lax"/>
   </xsd:complexType>

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

```
    </xsd:complexType>

    <xsd:complexType name="ECReportOutputSpec">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
             ECReportOutputSpec specifies how the final set of EPCs is to be reported
             with respect to type.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="extension" type="ale:ECReportOutputSpecExtension"
                         minOccurs="0"/>
            <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                     namespace="##other"/>
        </xsd:sequence>
        <xsd:attribute name="includeEPC" type="xsd:boolean" default="false"/>
        <xsd:attribute name="includeTag" type="xsd:boolean" default="false"/>
        <xsd:attribute name="includeRawHex" type="xsd:boolean" default="false"/>
        <xsd:attribute name="includeRawDecimal" type="xsd:boolean" default="false"/>
        <xsd:attribute name="includeCount" type="xsd:boolean" default="false"/>
    </xsd:complexType>

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


    <xsd:complexType name="ECReports">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
             ECReports is the output from an event cycle. The "meat" of an ECReports
             instance is the lists of count report instances and list report
             instances, each corresponding to an ECReportSpec instance in the event
             cycle's ECSpec. In addition to the reports themselves, ECReports contains
             a number of "header" fields that provide useful information about the
             event cycle.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="epcglobal:Document">
                <xsd:sequence>
                    <xsd:element name="reports" type="ale:ECReportList"/>
                    <xsd:element name="extension" type="ale:ECReportsExtension"
                                 minOccurs="0"/>
                    <xsd:element name="ECSpec" type="ale:ECSpec" minOccurs="0"/>
                    <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                             namespace="##other"/>
                </xsd:sequence>
                <xsd:attribute name="specName" type="xsd:string" use="required"/>
                <xsd:attribute name="date" type="xsd:dateTime" use="required"/>
                <xsd:attribute name="ALEID" type="xsd:string" use="required"/>
                <xsd:attribute name="totalMilliseconds" type="xsd:long" use="required"/>
                <xsd:attribute name="terminationCondition"
                               type="ale:ECTerminationCondition" use="required"/>
                 <xsd:attribute name="schemaURL" type="xsd:string" use="optional"/>
                <xsd:anyAttribute processContents="lax"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

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

```
    </xsd:complexType>


    <xsd:complexType name="ECReportSetSpec">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
             ECReportSetSpec specifies which set of EPCs is to be considered for
             filtering and output.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:attribute name="set" type="ale:ECReportSetEnum"/>
    </xsd:complexType>

    <xsd:simpleType name="ECReportSetEnum">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
             ECReportSetEnum 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.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:NCName">
            <xsd:enumeration value="CURRENT"/>
            <xsd:enumeration value="ADDITIONS"/>
            <xsd:enumeration value="DELETIONS"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="ECReportSpec">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
             A ReportSpec specifies one report to be returned from executing an event
             cycle. An ECSpec may contain one or more ECReportSpec instances.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="reportSet" type="ale:ECReportSetSpec"/>
            <xsd:element name="filterSpec" type="ale:ECFilterSpec" minOccurs="0"/>
            <xsd:element name="groupSpec" type="ale:ECGroupSpec" minOccurs="0"/>
            <xsd:element name="output" type="ale:ECReportOutputSpec"/>
            <xsd:element name="extension" type="ale:ECReportSpecExtension"
                         minOccurs="0"/>
            <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                     namespace="##other"/>
        </xsd:sequence>
        <xsd:attribute name="reportName" type="xsd:string" use="required"/>
        <xsd:attribute name="reportIfEmpty" type="xsd:boolean" default="false"/>
       <xsd:attribute name="reportOnlyOnChange" type="xsd:boolean" default="false"/>
        <xsd:anyAttribute processContents="lax"/>
    </xsd:complexType>

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


    <xsd:complexType name="ECReportSpecs">
        <xsd:sequence>
            <xsd:element name="reportSpec" type="ale:ECReportSpec"
                         maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ECSpec">
        <xsd:annotation>
```

```
            <xsd:documentation xml:lang="en">
             An ECSpec describes an event cycle and one or more reports that are to
             be generated from it. It contains a list of logical readers whose reader
             cycles are to be included in the event cycle, a specification of read
             cycle timing, a specification of how the boundaries of event cycles are
             to be determined, and list of specifications each of which describes a
             report to be generated from this event cycle.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="epcglobal:Document">
                <xsd:sequence>
                    <xsd:element name="logicalReaders" type="ale:ECLogicalReaders"/>
                    <xsd:element name="boundarySpec" type="ale:ECBoundarySpec"/>
                    <xsd:element name="reportSpecs" type="ale:ECReportSpecs"/>
                    <xsd:element name="extension" type="ale:ECSpecExtension"
                                 minOccurs="0"/>
                    <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"
                             namespace="##other"/>
                </xsd:sequence>
                <xsd:attribute name="includeSpecInReports" type="xsd:boolean"
                               default="false"/>
                <xsd:anyAttribute processContents="lax"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

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


    <xsd:simpleType name="ECTerminationCondition">
        <xsd:restriction base="xsd:NCName">
            <xsd:enumeration value="TRIGGER"/>
            <xsd:enumeration value="DURATION"/>
            <xsd:enumeration value="STABLE_SET"/>
            <xsd:enumeration value="UNREQUEST"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="ECTime">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
             An ECTime specifies a time duration in physical units.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
            <xsd:extension base="xsd:long">
                <xsd:attribute name="unit" type="ale:ECTimeUnit"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:simpleType name="ECTimeUnit">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
             ECTimeUnit represents the supported physical time unit: millisecond
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:NCName">
            <xsd:enumeration value="MS"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="ECTrigger">
```

```
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
             A trigger is a URI that is used to specify a start or stop trigger for
             an event cycle.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleContent>
            <xsd:extension base="xsd:string"/>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:schema>
```

## 10.3 ECSpec – Example (non-normative)

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

```
<?xml version="1.0" encoding="UTF-8"?>
<ale:ECSpec xmlns:ale="urn:epcglobal:ale:xsd:1"
            xmlns:epcglobal="urn:epcglobal:xsd:1"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:epcglobal:ale:xsd:1 Ale.xsd"
            schemaVersion="1.0"
            creationDate="2003-08-06T10:54:06.444-05:00">
    <logicalReaders>
        <logicalReader>dock_1a</logicalReader>
        <logicalReader>dock_1b</logicalReader>
    </logicalReaders>
    <boundarySpec>
        <startTrigger>http://sample.com/trigger1</startTrigger>
        <repeatPeriod unit="MS">20000</repeatPeriod>
        <stopTrigger>http://sample.com/trigger2</stopTrigger>
        <duration unit="MS">3000</duration>
    </boundarySpec>
    <reportSpecs>
        <reportSpec reportName="report1">
            <reportSet set="CURRENT"/>
            <output includeTag="true"/>
        </reportSpec>
        <reportSpec reportName="report2">
            <reportSet set="ADDITIONS"/>
            <output includeCount="true"/>
        </reportSpec>
        <reportSpec reportName="report3">
            <reportSet set="DELETIONS"/>
            <groupSpec>
               <pattern>urn:epc:pat:sgtin-64:X.X.X.*</pattern>
            </groupSpec>
            <output includeCount="true"/>
        </reportSpec>
    </reportSpecs>
</ale:ECSpec>
```

## 10.4 ECReports – Example (non-normative)

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

```
<?xml version="1.0" encoding="UTF-8"?>
<ale:ECReports xmlns:ale="urn:epcglobal:ale:xsd:1"
               xmlns:epcglobal="urn:epcglobal:xsd:1"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:epcglobal:ale:xsd:1 Ale.xsd"
               schemaVersion="1.0"
               creationDate="2003-08-06T10:54:06.444-05:00"
               specName="EventCycle1"
               date="2003-08-06T10:54:06.444-05:00"
               ALEID="Edge34"
               totalMilliseconds="3034"
               terminationCondition="DURATION">
```

```
    <reports>
        <report reportName="report1">
            <group>
                <groupList>
                    <member><tag>urn:epc:tag:gid-96:10.50.1000</tag></member>
                    <member><tag>urn:epc:tag:gid-96:10.50.1001</tag></member>
                </groupList>
            </group>
        </report>
        <report reportName="report2">
            <group><groupCount><count>6847</count></groupCount></group>
        </report>
        <report reportName="report3">
            <group name="urn:epc:pat:sgtin-64:3.0037000.12345.*">
               <groupCount><count>2</count></groupCount>
            </group>
            <group name="urn:epc:pat:sgtin-64:3.0037000.55555.*">
               <groupCount><count>3</count></groupCount>
            </group>
            <group>
               <groupCount><count>6842</count></groupCount>
            </group>
        </report>
    </reports>
</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].

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ALESERVICE DEFINITIONS -->
<wsdl:definitions
    targetNamespace="urn:epcglobal:ale:wsdl:1"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:impl="urn:epcglobal:ale:wsdl:1"
    xmlns:ale="urn:epcglobal:ale:xsd:1"
    xmlns:epcglobal="urn:epcglobal:xsd:1"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <wsdl:documentation>
      <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></epcglobal:specification>

       This WSDL document describes the types, messages, operations, and
       bindings for the ALEService.
    </wsdl:documentation>

    <!-- ALESERVICE TYPES -->
    <wsdl:types>
        <xsd:schema targetNamespace="urn:epcglobal:ale:wsdl:1"
                    xmlns:impl="urn:epcglobal:ale:wsdl:1"
```

```
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:import namespace="urn:epcglobal:ale:xsd:1"
            schemaLocation="./ALE.xsd"/>

<!-- ALESERVICE MESSAGE WRAPPERS -->
<xsd:element name="Define" type="impl:Define"/>
<xsd:complexType name="Define">
   <xsd:sequence>
      <xsd:element name="specName" type="xsd:string"/>
      <xsd:element name="spec" type="ale:ECSpec"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="Undefine" type="impl:Undefine"/>
 <xsd:complexType name="Undefine">
   <xsd:sequence>
      <xsd:element name="specName" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>

<xsd:element name="GetECSpec" type="impl:GetECSpec"/>
 <xsd:complexType name="GetECSpec">
   <xsd:sequence>
      <xsd:element name="specName" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>
<xsd:element name="GetECSpecResult" type="ale:ECSpec"/>

<xsd:element name="GetECSpecNames" type="impl:EmptyParms"/>
<xsd:element name="GetECSpecNamesResult" type="impl:ArrayOfString"/>

<xsd:element name="Subscribe" type="impl:Subscribe"/>
 <xsd:complexType name="Subscribe">
   <xsd:sequence>
      <xsd:element name="specName" type="xsd:string"/>
      <xsd:element name="notificationURI" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>

<xsd:element name="Unsubscribe" type="impl:Unsubscribe"/>
 <xsd:complexType name="Unsubscribe">
   <xsd:sequence>
      <xsd:element name="specName" type="xsd:string"/>
      <xsd:element name="notificationURI" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>

<xsd:element name="Poll" type="impl:Poll"/>
 <xsd:complexType name="Poll">
   <xsd:sequence>
      <xsd:element name="specName" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>
<xsd:element name="PollResult" type="ale:ECReports"/>

<xsd:element name="Immediate" type="impl:Immediate"/>
 <xsd:complexType name="Immediate">
   <xsd:sequence>
      <xsd:element name="spec" type="ale:ECSpec"/>
   </xsd:sequence>
 </xsd:complexType>
<xsd:element name="ImmediateResult" type="ale:ECReports"/>

<xsd:element name="GetSubscribers" type="impl:GetSubscribers"/>
 <xsd:complexType name="GetSubscribers">
   <xsd:sequence>
      <xsd:element name="specName" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>
<xsd:element name="GetSubscribersResult" type="impl:ArrayOfString"/>
```

```
                <xsd:element name="GetStandardVersion" type="impl:EmptyParms"/>
                <xsd:element name="GetStandardVersionResult" type="xsd:string"/>

                <xsd:element name="GetVendorVersion" type="impl:EmptyParms"/>
                <xsd:element name="GetVendorVersionResult" type="xsd:string"/>

                <xsd:element name="VoidHolder" type="impl:VoidHolder"/>
                 <xsd:complexType name="VoidHolder">
                   <xsd:sequence>
                   </xsd:sequence>
                 </xsd:complexType>

                <xsd:complexType name="EmptyParms"/>

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

                 <!-- ALE EXCEPTIONS -->
                  <xsd:element name="ALEException" type="impl:ALEException"/>
                 <xsd:complexType name="ALEException">
                   <xsd:sequence>
                      <xsd:element name="reason" type="xsd:string"/>
                   </xsd:sequence>
                 </xsd:complexType>

                  <xsd:element name="SecurityException"
                               type="impl:SecurityException"/>
                 <xsd:complexType name="SecurityException">
                   <xsd:complexContent>
                      <xsd:extension base="impl:ALEException">
                         <xsd:sequence/>
                      </xsd:extension>
                   </xsd:complexContent>
                 </xsd:complexType>

                  <xsd:element name="DuplicateNameException"
                               type="impl:DuplicateNameException"/>
                 <xsd:complexType name="DuplicateNameException">
                   <xsd:complexContent>
                      <xsd:extension base="impl:ALEException">
                         <xsd:sequence/>
                      </xsd:extension>
                   </xsd:complexContent>
                 </xsd:complexType>

                  <xsd:element name="ECSpecValidationException"
                               type="impl:ECSpecValidationException"/>
                 <xsd:complexType name="ECSpecValidationException">
                   <xsd:complexContent>
                      <xsd:extension base="impl:ALEException">
                         <xsd:sequence/>
                      </xsd:extension>
                   </xsd:complexContent>
                 </xsd:complexType>

                  <xsd:element name="InvalidURIException" type="impl:InvalidURIException"/>
                 <xsd:complexType name="InvalidURIException">
                   <xsd:complexContent>
                      <xsd:extension base="impl:ALEException">
                         <xsd:sequence/>
                      </xsd:extension>
                   </xsd:complexContent>
                 </xsd:complexType>

                  <xsd:element name="NoSuchNameException" type="impl:NoSuchNameException"/>
```

```
            <xsd:complexType name="NoSuchNameException">
               <xsd:complexContent>
                  <xsd:extension base="impl:ALEException">
                     <xsd:sequence/>
                  </xsd:extension>
               </xsd:complexContent>
            </xsd:complexType>

             <xsd:element name="NoSuchSubscriberException"
                          type="impl:NoSuchSubscriberException"/>
            <xsd:complexType name="NoSuchSubscriberException">
               <xsd:complexContent>
                  <xsd:extension base="impl:ALEException">
                     <xsd:sequence/>
                  </xsd:extension>
               </xsd:complexContent>
            </xsd:complexType>

             <xsd:element name="DuplicateSubscriptionException"
                          type="impl:DuplicateSubscriptionException"/>
            <xsd:complexType name="DuplicateSubscriptionException">
               <xsd:complexContent>
                  <xsd:extension base="impl:ALEException">
                     <xsd:sequence/>
                  </xsd:extension>
               </xsd:complexContent>
            </xsd:complexType>

             <xsd:element name="ImplementationException"
                          type="impl:ImplementationException"/>
            <xsd:complexType name="ImplementationException">
               <xsd:complexContent>
                  <xsd:extension base="impl:ALEException">
                     <xsd:sequence>
                        <xsd:element name="severity"
                                     type="impl:ImplementationExceptionSeverity"/>
                    </xsd:sequence>
                  </xsd:extension>
               </xsd:complexContent>
            </xsd:complexType>

           <xsd:simpleType name="ImplementationExceptionSeverity">
               <xsd:restriction base="xsd:NCName">
                   <xsd:enumeration value="ERROR"/>
                   <xsd:enumeration value="SEVERE"/>
               </xsd:restriction>
           </xsd:simpleType>

      </xsd:schema>
  </wsdl:types>

  <!-- ALESERVICE MESSAGES -->
  <wsdl:message name="defineRequest">
      <wsdl:part name="parms" element="impl:Define"/>
  </wsdl:message>
  <wsdl:message name="defineResponse">
      <wsdl:part name="defineReturn" element="impl:VoidHolder"/>
  </wsdl:message>

  <wsdl:message name="undefineRequest">
      <wsdl:part name="parms" element="impl:Undefine"/>
  </wsdl:message>
  <wsdl:message name="undefineResponse">
      <wsdl:part name="undefineReturn" element="impl:VoidHolder"/>
  </wsdl:message>

  <wsdl:message name="getECSpecRequest">
      <wsdl:part name="parms" element="impl:GetECSpec"/>
  </wsdl:message>
  <wsdl:message name="getECSpecResponse">
```

```
        <wsdl:part name="getECSpecReturn" element="impl:GetECSpecResult"/>
    </wsdl:message>

    <wsdl:message name="getECSpecNamesRequest">
        <wsdl:part name="parms" element="impl:GetECSpecNames"/>
    </wsdl:message>
    <wsdl:message name="getECSpecNamesResponse">
        <wsdl:part name="getECSpecNamesReturn" element="impl:GetECSpecNamesResult"/>
    </wsdl:message>

    <wsdl:message name="subscribeRequest">
        <wsdl:part name="parms" element="impl:Subscribe"/>
    </wsdl:message>
    <wsdl:message name="subscribeResponse">
        <wsdl:part name="subscribeReturn" element="impl:VoidHolder"/>
    </wsdl:message>

    <wsdl:message name="unsubscribeRequest">
       <wsdl:part name="parms" element="impl:Unsubscribe"/>
    </wsdl:message>
    <wsdl:message name="unsubscribeResponse">
        <wsdl:part name="unsubscribeReturn" element="impl:VoidHolder"/>
    </wsdl:message>

    <wsdl:message name="pollRequest">
        <wsdl:part name="parms" element="impl:Poll"/>
    </wsdl:message>
    <wsdl:message name="pollResponse">
        <wsdl:part name="pollReturn" element="impl:PollResult"/>
    </wsdl:message>

    <wsdl:message name="immediateRequest">
        <wsdl:part name="parms" element="impl:Immediate"/>
    </wsdl:message>
    <wsdl:message name="immediateResponse">
        <wsdl:part name="immediateReturn" element="impl:ImmediateResult"/>
    </wsdl:message>

    <wsdl:message name="getSubscribersRequest">
        <wsdl:part name="parms" element="impl:GetSubscribers"/>
    </wsdl:message>
    <wsdl:message name="getSubscribersResponse">
        <wsdl:part name="getSubscribersReturn" element="impl:GetSubscribersResult"/>
    </wsdl:message>

    <wsdl:message name="getStandardVersionRequest">
        <wsdl:part name="parms" element="impl:GetStandardVersion"/>
    </wsdl:message>
    <wsdl:message name="getStandardVersionResponse">
        <wsdl:part name="getStandardVersionReturn"
element="impl:GetStandardVersionResult"/>
    </wsdl:message>


    <wsdl:message name="getVendorVersionRequest">
        <wsdl:part name="parms" element="impl:GetVendorVersion"/>
    </wsdl:message>
    <wsdl:message name="getVendorVersionResponse">
        <wsdl:part name="getVendorVersionReturn" element="impl:GetVendorVersionResult"/>
    </wsdl:message>

    <!-- ALESERVICE FAULT EXCEPTIONS -->
    <wsdl:message name="DuplicateNameExceptionResponse">
        <wsdl:part name="fault" element="impl:DuplicateNameException"/>
    </wsdl:message>
    <wsdl:message name="ECSpecValidationExceptionResponse">
        <wsdl:part name="fault" element="impl:ECSpecValidationException"/>
    </wsdl:message>
    <wsdl:message name="InvalidURIExceptionResponse">
        <wsdl:part name="fault" element="impl:InvalidURIException"/>
```

```
    </wsdl:message>
    <wsdl:message name="NoSuchNameExceptionResponse">
        <wsdl:part name="fault" element="impl:NoSuchNameException"/>
    </wsdl:message>
    <wsdl:message name="NoSuchSubscriberExceptionResponse">
        <wsdl:part name="fault" element="impl:NoSuchSubscriberException"/>
    </wsdl:message>
    <wsdl:message name="DuplicateSubscriptionExceptionResponse">
        <wsdl:part name="fault" element="impl:DuplicateSubscriptionException"/>
    </wsdl:message>
    <wsdl:message name="ImplementationExceptionResponse">
        <wsdl:part name="fault" element="impl:ImplementationException"/>
    </wsdl:message>
    <wsdl:message name="SecurityExceptionResponse">
        <wsdl:part name="fault" element="impl:SecurityException"/>
    </wsdl:message>

    <!-- ALESERVICE PORTTYPE -->
    <wsdl:portType name="ALEServicePortType">
        <wsdl:operation name="define">
            <wsdl:input message="impl:defineRequest" name="defineRequest"/>
            <wsdl:output message="impl:defineResponse" name="defineResponse"/>
            <wsdl:fault message="impl:DuplicateNameExceptionResponse"
                        name="DuplicateNameExceptionFault"/>
            <wsdl:fault message="impl:ECSpecValidationExceptionResponse"
                        name="ECSpecValidationExceptionFault"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="undefine">
            <wsdl:input message="impl:undefineRequest" name="undefineRequest"/>
            <wsdl:output message="impl:undefineResponse" name="undefineResponse"/>
            <wsdl:fault message="impl:NoSuchNameExceptionResponse"
                        name="NoSuchNameExceptionFault"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="getECSpec">
            <wsdl:input message="impl:getECSpecRequest" name="getECSpecRequest"/>
            <wsdl:output message="impl:getECSpecResponse" name="getECSpecResponse"/>
            <wsdl:fault message="impl:NoSuchNameExceptionResponse"
                        name="NoSuchNameExceptionFault"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="getECSpecNames">
            <wsdl:input message="impl:getECSpecNamesRequest"
                        name="getECSpecNamesRequest"/>
            <wsdl:output message="impl:getECSpecNamesResponse"
                         name="getECSpecNamesResponse"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="subscribe">
            <wsdl:input message="impl:subscribeRequest" name="subscribeRequest"/>
            <wsdl:output message="impl:subscribeResponse" name="subscribeResponse"/>
            <wsdl:fault message="impl:NoSuchNameExceptionResponse"
                        name="NoSuchNameExceptionFault"/>
```

```
            <wsdl:fault message="impl:InvalidURIExceptionResponse"
                        name="InvalidURIExceptionFault"/>
            <wsdl:fault message="impl:DuplicateSubscriptionExceptionResponse"
                        name="DuplicateSubscriptionExceptionFault"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="unsubscribe">
            <wsdl:input message="impl:unsubscribeRequest" name="unsubscribeRequest"/>
            <wsdl:output message="impl:unsubscribeResponse" name="unsubscribeResponse"/>
            <wsdl:fault message="impl:NoSuchNameExceptionResponse"
                        name="NoSuchNameExceptionFault"/>
            <wsdl:fault message="impl:NoSuchSubscriberExceptionResponse"
                        name="NoSuchSubscriberExceptionFault"/>
            <wsdl:fault message="impl:InvalidURIExceptionResponse"
                        name="InvalidURIExceptionFault"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="poll">
            <wsdl:input message="impl:pollRequest" name="pollRequest"/>
            <wsdl:output message="impl:pollResponse" name="pollResponse"/>
            <wsdl:fault message="impl:NoSuchNameExceptionResponse"
                        name="NoSuchNameExceptionFault"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="immediate">
            <wsdl:input message="impl:immediateRequest" name="immediateRequest"/>
            <wsdl:output message="impl:immediateResponse" name="immediateResponse"/>
            <wsdl:fault message="impl:ECSpecValidationExceptionResponse"
                        name="ECSpecValidationExceptionFault"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="getSubscribers">
            <wsdl:input message="impl:getSubscribersRequest"
                        name="getSubscribersRequest"/>
            <wsdl:output message="impl:getSubscribersResponse"
                         name="getSubscribersResponse"/>
            <wsdl:fault message="impl:NoSuchNameExceptionResponse"
                        name="NoSuchNameExceptionFault"/>
            <wsdl:fault message="impl:SecurityExceptionResponse"
                        name="SecurityExceptionFault"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
                        name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="getStandardVersion">
            <wsdl:input message="impl:getStandardVersionRequest"
name="getStandardVersionRequest"/>
            <wsdl:output message="impl:getStandardVersionResponse"
name="getStandardVersionResponse"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
name="ImplementationExceptionFault"/>
        </wsdl:operation>

        <wsdl:operation name="getVendorVersion">
```

```
            <wsdl:input message="impl:getVendorVersionRequest"
name="getVendorVersionRequest"/>
            <wsdl:output message="impl:getVendorVersionResponse"
name="getVendorVersionResponse"/>
            <wsdl:fault message="impl:ImplementationExceptionResponse"
name="ImplementationExceptionFault"/>
        </wsdl:operation>            </wsdl:portType>

    <!-- ALESERVICE BINDING -->
    <wsdl:binding name="ALEServiceBinding" type="impl:ALEServicePortType">
        <wsdlsoap:binding style="document"
                          transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="define">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="defineRequest">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:input>
            <wsdl:output name="defineResponse">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:output>
            <wsdl:fault name="DuplicateNameExceptionFault">
                <wsdlsoap:fault
                    name="DuplicateNameExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="ECSpecValidationExceptionFault">
                <wsdlsoap:fault
                    name="ECSpecValidationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="SecurityExceptionFault">
                <wsdlsoap:fault
                    name="SecurityExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="ImplementationExceptionFault">
                <wsdlsoap:fault
                    name="ImplementationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
        </wsdl:operation>

        <wsdl:operation name="undefine">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="undefineRequest">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:input>
            <wsdl:output name="undefineResponse">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:output>
            <wsdl:fault name="NoSuchNameExceptionFault">
                <wsdlsoap:fault
                    name="NoSuchNameExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="SecurityExceptionFault">
                <wsdlsoap:fault
                    name="SecurityExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="ImplementationExceptionFault">
                <wsdlsoap:fault
                    name="ImplementationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
        </wsdl:operation>
```

```
    <wsdl:operation name="getECSpec">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="getECSpecRequest">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:input>
        <wsdl:output name="getECSpecResponse">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:output>
        <wsdl:fault name="NoSuchNameExceptionFault">
            <wsdlsoap:fault
                name="NoSuchNameExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="SecurityExceptionFault">
            <wsdlsoap:fault
                name="SecurityExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="ImplementationExceptionFault">
            <wsdlsoap:fault
                name="ImplementationExceptionFault"
                use="literal"/>
        </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="getECSpecNames">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="getECSpecNamesRequest">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:input>
        <wsdl:output name="getECSpecNamesResponse">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:output>
        <wsdl:fault name="SecurityExceptionFault">
            <wsdlsoap:fault
                name="SecurityExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="ImplementationExceptionFault">
            <wsdlsoap:fault
                name="ImplementationExceptionFault"
                use="literal"/>
        </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="subscribe">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="subscribeRequest">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:input>
        <wsdl:output name="subscribeResponse">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:output>
        <wsdl:fault name="NoSuchNameExceptionFault">
            <wsdlsoap:fault
                name="NoSuchNameExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="InvalidURIExceptionFault">
            <wsdlsoap:fault
                name="InvalidURIExceptionFault"
                use="literal"/>
        </wsdl:fault>
```

```
        <wsdl:fault name="DuplicateSubscriptionExceptionFault">
            <wsdlsoap:fault
                name="DuplicateSubscriptionExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="SecurityExceptionFault">
            <wsdlsoap:fault
                name="SecurityExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="ImplementationExceptionFault">
            <wsdlsoap:fault
                name="ImplementationExceptionFault"
                use="literal"/>
        </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="unsubscribe">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="unsubscribeRequest">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:input>
        <wsdl:output name="unsubscribeResponse">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:output>
        <wsdl:fault name="NoSuchNameExceptionFault">
            <wsdlsoap:fault
                name="NoSuchNameExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="NoSuchSubscriberExceptionFault">
            <wsdlsoap:fault
                name="NoSuchSubscriberExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="InvalidURIExceptionFault">
            <wsdlsoap:fault
                name="InvalidURIExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="SecurityExceptionFault">
            <wsdlsoap:fault
                name="SecurityExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="ImplementationExceptionFault">
            <wsdlsoap:fault
                name="ImplementationExceptionFault"
                use="literal"/>
        </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="poll">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="pollRequest">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:input>
        <wsdl:output name="pollResponse">
            <wsdlsoap:body
                use="literal"/>
        </wsdl:output>
        <wsdl:fault name="NoSuchNameExceptionFault">
            <wsdlsoap:fault
                name="NoSuchNameExceptionFault"
                use="literal"/>
        </wsdl:fault>
        <wsdl:fault name="SecurityExceptionFault">
```

```
                <wsdlsoap:fault
                    name="SecurityExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="ImplementationExceptionFault">
                <wsdlsoap:fault
                    name="ImplementationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
        </wsdl:operation>

        <wsdl:operation name="immediate">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="immediateRequest">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:input>
            <wsdl:output name="immediateResponse">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:output>
            <wsdl:fault name="ECSpecValidationExceptionFault">
                <wsdlsoap:fault
                    name="ECSpecValidationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="SecurityExceptionFault">
                <wsdlsoap:fault
                    name="SecurityExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="ImplementationExceptionFault">
                <wsdlsoap:fault
                    name="ImplementationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
        </wsdl:operation>

        <wsdl:operation name="getSubscribers">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="getSubscribersRequest">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:input>
            <wsdl:output name="getSubscribersResponse">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:output>
            <wsdl:fault name="NoSuchNameExceptionFault">
                <wsdlsoap:fault
                    name="NoSuchNameExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="SecurityExceptionFault">
                <wsdlsoap:fault
                    name="SecurityExceptionFault"
                    use="literal"/>
            </wsdl:fault>
            <wsdl:fault name="ImplementationExceptionFault">
                <wsdlsoap:fault
                    name="ImplementationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
        </wsdl:operation>

        <wsdl:operation name="getStandardVersion">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="getStandardVersionRequest">
                <wsdlsoap:body
                    use="literal"/>
```

```
            </wsdl:input>
            <wsdl:output name="getStandardVersionResponse">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:output>
            <wsdl:fault name="ImplementationExceptionFault">
                <wsdlsoap:fault
                    name="ImplementationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
        </wsdl:operation>

        <wsdl:operation name="getVendorVersion">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="getVendorVersionRequest">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:input>
            <wsdl:output name="getVendorVersionResponse">
                <wsdlsoap:body
                    use="literal"/>
            </wsdl:output>
            <wsdl:fault name="ImplementationExceptionFault">
                <wsdlsoap:fault
                    name="ImplementationExceptionFault"
                    use="literal"/>
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>

    <!-- ALESERVICE -->
    <wsdl:service name="ALEService">
        <wsdl:port binding="impl:ALEServiceBinding" name="ALEServicePort">
            <!-- The value of the location attribute below is an example only;
                 Implementations are free to choose any appropriate URL. -->
            <wsdlsoap:address
                location="http://localhost:6060/axis/services/ALEService"/>
        </wsdl:port>
    </wsdl:service>

</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 = pallet/case GTIN |
| 2a (retail OOS) | Periodic | Additions & Deletions | Item | Group cardinality, G = item GTIN |
| 2b (retail OOS) | Periodic | Complete | Item | Group cardinality, G = 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 = item GTIN |
| 6 (forklift) | Single | Complete | Slot | Membership (EPC) |

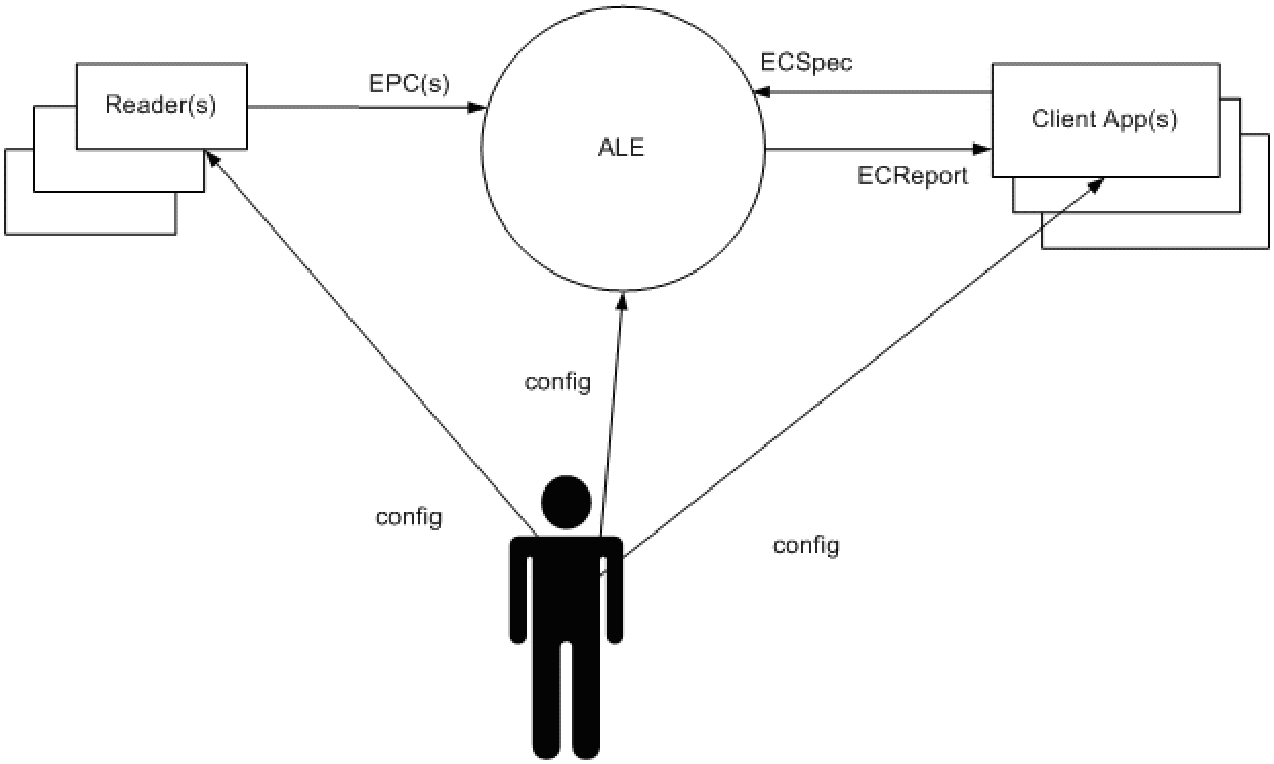
# 13 ALE Scenarios (non-normative)

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

## 13.1 ALE Context

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




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

- `define`, `undefine`
- `subscribe`, `unsubscribe`
- `poll`
- `immediate`
- `getECSpecNames`, `getECSpec`

These methods are defined normatively in Section 8.1.

## 13.2 Scenarios

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

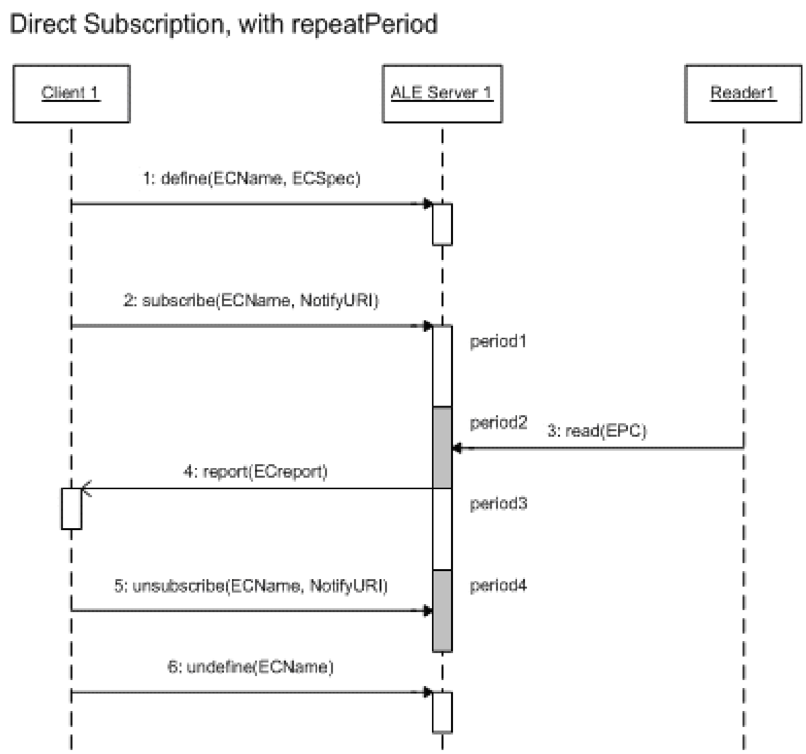
1. Defining Subscribe ECName, ECSpec
    a. Direct Subscription. Defined by A, Report to: A
    b. Indirect Subscription Defined by A, Report to: B
2. Poll(ECName)

3. Immediate(ECSpec)
4. Operation Errors
5. System Errors

### 13.2.1 Scenario 1a: Direct Subscription

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




#### 13.2.1.1 Assumptions

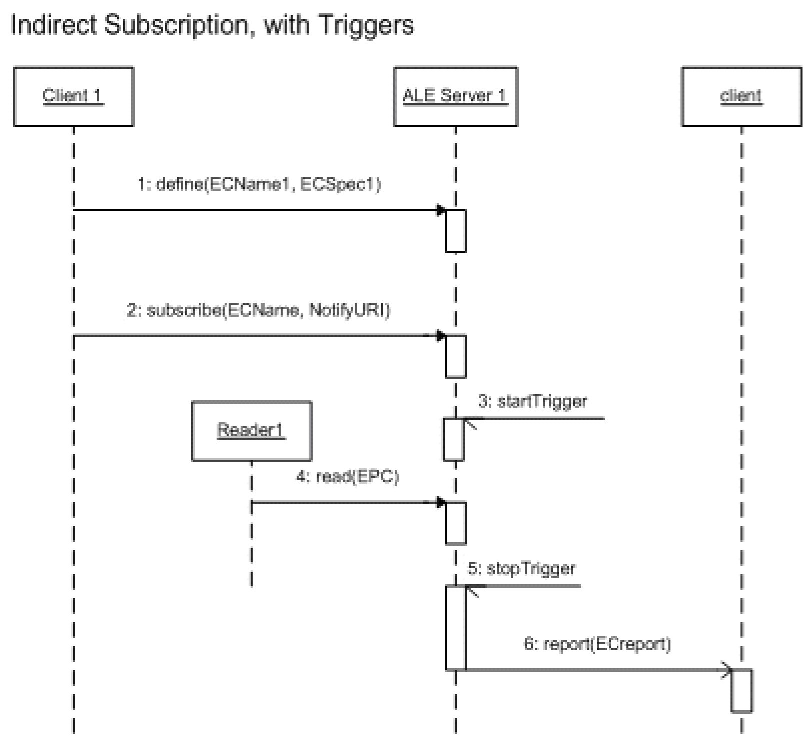
1. All discovery, configuration, and initialization required has already been performed.
2. The ALE layer is implemented as a distinct software component.
3. `ECSpec` boundary condition specified using: `repeatPeriod`
4. `ECFilterSpec includePatterns` includes the EPC(s) illustrated in this scenario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

#### 13.2.1.2 Description

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

### 13.2.2 Scenario 1b: Indirect Subscription

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




#### 13.2.2.1 Assumptions

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

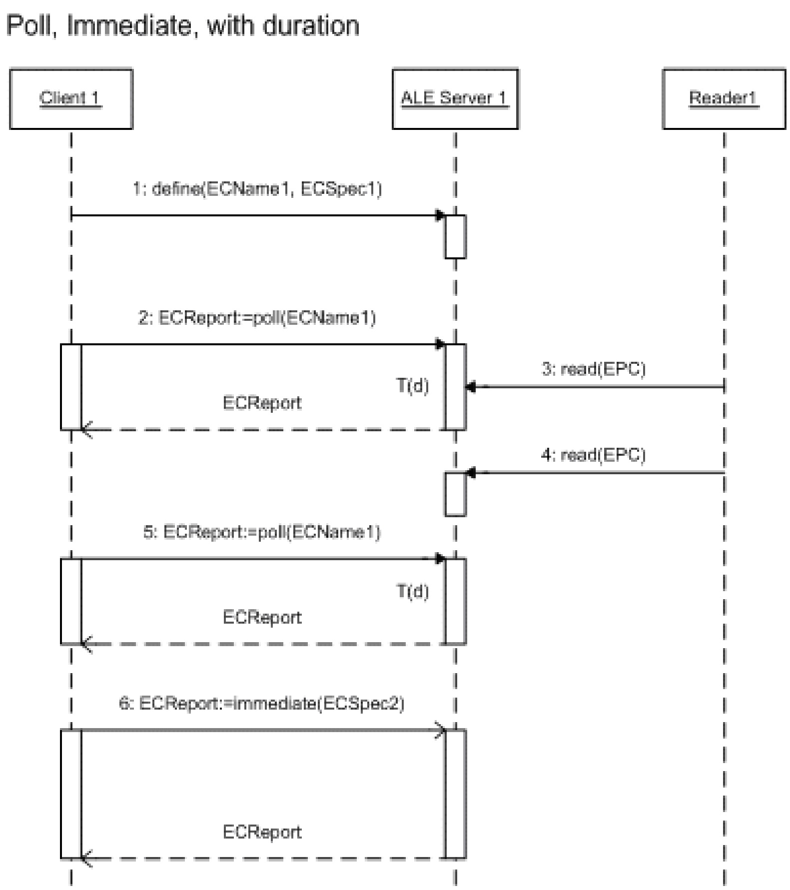
#### 13.2.2.2 Description

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

5. The stop trigger is received. The `ECSpec` transitions to the "Requested" state.
6. The `ECReports` is generated and sent asynchronously to the observer.

### 13.2.3 Scenario 2, 3: Poll, Immediate

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




#### 13.2.3.1 Assumptions

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

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

#### 13.2.3.2 Description

1. The ALE client calls the `define` method of the ALE interface. The `ECSpec` contains a valid `duration` as the boundary specification – though any valid boundary conditions could be specified. The `ECReportSpec` and `ECFilterSpec` contained within the `ECSpec` are defined to include the EPC data sent later in steps 3 and 4. At this point the `ECSpec` is considered "Unrequested."
2. The ALE client calls the `poll` method of the ALE interface, naming the `ECSpec` previously defined in Step 1. At this point the `ECSpec` is transitioned to the "Active" state, and the event cycle begins for the duration specified in the `ECSpec`. During the duration of the event cycle the ALE client is blocked waiting for a response to the `poll` method.
3. An EPC which meets the filter conditions of the `ECSpec` is received during the event cycle. At the end of the event cycle, the `ECReports` is generated and returned to the ALE client as the response to the `poll` method. At this point the `ECSpec` transitions to the "Unrequested" state.
4. An EPC that meets the filter conditions of the `ECSpec` is reported to the ALE layer, but since there is no "Active" `ECSpec`, this EPC will not be reported.
5. The ALE client invokes the `poll` method of the ALE interface a second time. This is similar to the process described above in Steps 2 and 3, but since no EPC is received, no EPC data is returned in the `ECReports`.
6. Later, the ALE client calls the `immediate` method of the ALE interface. This is very similar to the use of `poll`, except that when the client calls `immediate` it provides the `ECSpec` as part of the method call, as opposed to referring to a previously defined `ECSpec`. Since a new `ECSpec` is provided with the `immediate` method, it can contain any valid combination of parameters and report options.

# 14 Glossary (non-normative)

This section provides a non-normative summary of terms used within this specification. For normative definitions of these terms, please consult the relevant sections of the 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 `immediate` or `poll` 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 `subscribe` 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; *e.g.*, DockDoor42. |

# 15 References

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

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

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

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

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

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

[TDS1.1] EPCglobal, “EPC Tag Data Standards Version 1.1 Rev.1.24,” EPCglobal Standard Specification, April 2004, http://www.epcglobalinc.org/standards_technology/EPCTagDataSpecification11rev124.pdf.

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

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

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

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

[XSD2] P. Biron, A. Malhotra, “XML Schema Part 2: Datatypes,” W3C Recommendation, May 2001, http://www.w3.org/TR/xmlschema-2/.

[XMLVersioning] D. Orchard, “Versioning XML Vocabularies,” December 2003, http://www.xml.com/pub/a/2003/12/03/versioning.html.