The Application Level Events (ALE) Specification, Version 1.1.1 Part I: Core Specification

EPCglobal Ratified Standard with Fixed Errata

13 March 2009

Previous Version: 1.1

Disclaimer

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

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

Abstract

This document specifies an interface through which clients may interact with filtered, consolidated EPC data and related 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 interface described herein, and the functionality it implies, is called "Application Level Events," or ALE.

This ALE 1.1 specification is a backward-compatible follow on specification to the ALE 1.0 specification, ratified by EPCglobal in September 2005. The ALE 1.0 specification provided only an interface for reading data (not writing), and only provided access to EPC data. The present ALE 1.1 specification expands upon ALE 1.0 to address writing as well as reading, and both the reading and writing aspects address not only EPC data but also other data that may be present on EPC data carriers. In particular, the ALE 1.1 specification is designed to provide full access to the functionality of the EPCglobal UHF Class 1 Gen 2 Air Interface ("Gen2") specification, when interacting with Gen2 RFID Tags. This includes reading and writing all memory banks, as well as exercising specific operations such as "lock" and "kill." In ALE 1.1, additional tag types may easily be accomodated in the future. In addition to providing reading and writing functionality, the ALE 1.1 specification also provides new interfaces for defining tag memory fields, for managing the naming of data source names ("logical readers"), and for securing the use of the APIs.

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 five means:

- It provides a means for clients to specify, in a high-level, declarative way, what data they are interested in or what operations they want performed, 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 or other devices 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 data and results from carrying out operations that is largely independent of where the data originated or how it was processed.
- It abstracts the channels through which data carriers are accessed 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 interact with 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.

- It abstracts the addressing of information stored on Tags and other data carriers into a higher-level notion of named, typed “fields,” hiding from clients the details of how a particular data element is encoded into a bit-level representation and stored at a particular address within a data carrier’s memory. This allows application logic to remain invariant despite differences between the memory organization of different data carriers (for example, differences between Gen 1 and Gen 2 RFID Tags), and also shields application logic from having to understand complex layout or data parsing rules.
- It provides a security mechanism so that administrators may choose which operations a given application may perform, as a policy that is decoupled from application logic itself.

This Part I specifies at an abstract level all interfaces that are part of the ALE specification, using UML notation. Part II of the specification [ALE1.1Part2] specifies XML-based wire protocol bindings of the interfaces, including XSD schemas for all data types, WS-I compliant WSDL definitions of SOAP bindings of the service interfaces, and several XML-based bindings of callback interfaces used in certain modes of reading and writing data. Implementations may provide additional bindings of the API, including bindings to particular programming languages.

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 fixes errata found in version 1.1 of ALE that was ratified on February 27, 2008. The Technical Steering Committeee (TSC) approved the errata fixes in the 1.1.1 version on March 13, 2009.

Comments on this document should be sent to the EPCglobal Software Action Group Filtering and Collection 1.1 Working Group mailing list sag_fc1_1_wg@lists.epcglobalinc.org.

Errata Fixed in ALE 1.1.1

The following table summarizes errata in ALE 1.1 that are fixed in ALE 1.1.1. All fixed errata in ALE 1.1.1 apply to Part I of the specification; there are no changes to Part II. For a comparison between ALE 1.1 and ALE 1.0, please see Section 16.

Section	Place	Old Text	Change
5.4.2	2nd bullet in 2nd bulleted list	"…it causes the tag to be omitted from the event cycle"	Changed to "…it is treated as a failure to match the pattern; that is, it causes the Tag to fail an INCLUDE filter or pass an EXCLUDE filter."
5.4.2	1st bullet in 3rd bulleted list	"…it causes the tag to be omitted from the event cycle"	Changed to "…it is treated as a failure to match the pattern; that is, it causes the Tag to fail an INCLUDE filter or pass an EXCLUDE filter."
5.4.3	2nd bullet in 1st bulleted list	"…it causes the tag to be omitted from the event cycle"	Changed to "…it is treated as a failure to match the pattern; that is, it causes the Tag to fail an INCLUDE filter or pass an EXCLUDE filter."
5.4.3	1st bullet in 2nd bulleted list	"…it causes the tag to be omitted from the event cycle"	Changed to "…it is treated as a failure to match the pattern; that is, it causes the Tag to fail an INCLUDE filter or pass an EXCLUDE filter."
5.4.3	2nd bullet in 2nd bulleted list	"`FIELD_NOT_FOUND_ERROR`"	Changed to "`OP_NOT_POSSIBLE_ERROR`"
6.2.1	2nd paragraph		Two parenthesized comments are added.

Section	Place	Old Text	Change
8.2	Last bullet following Table 31		Added sentence: "An implementation SHALL NOT, however, raise the exception if `primaryKeyFields` is omitted or its value is a list consisting of the single element `epc`."
8.2.14	20th bullet		Added sentence: "An implementation SHALL NOT, however, raise the exception if `primaryKeyFields` is omitted or its value is a list consisting of the single element `epc`."
9.3.5	Last row of Table 64, `dataSpec` column	"The lock action to be performed."	Changed to: "A LITERAL `dataSpec` whose value specifies the lock action to be performed."
9.3.5.2.2	3rd paragraph	"…in the EPC/UII memory bank."	Changed to: "…in the User memory bank."
9.5.3	First row of Table 82, Description column		Added to end of first sentence: "…which field must be numeric."
9.5.3	Non-normative note		Added text to end of non-normative note, beginning "Note that wildcard fields must be numeric…" and continuing through the end of the note.
17	[Gen2] bibliography entry		Updated to refer to Version 1.1.0 of the Gen2 specification.

Table of Contents

List of Tables

1 Introduction

This document specifies an interface through which clients may interact with filtered, consolidated EPC data and related 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.

Broadly speaking, client interactions with EPC data can be divided into *reading* activity and *writing* activity. For reading activity, the processing done between the physical data sources and client applications typically involves: (1) *receiving* EPCs and related data from one or more data sources such as RFID readers; (2) *accumulating* data over intervals of time, *filtering* to eliminate duplicate data and data that are not of interest, and *counting* and *grouping* data to reduce the volume of data; and (3) *reporting* in various forms. For writing activity, the processing typically involves: (1) *isolating* ("singulating") individual data carriers such as RFID Tags through one or more channels such as RFID readers; (2) *operating* upon the data carriers by writing data, reading data, or performing other operations; and (3) *reporting* in various forms. The interface described herein, and the functionality it implies, is called "Application Level Events," or ALE.

The ALE 1.0 specification [ALE1.0], ratified by EPCglobal in September 2005, was the first specification at this level of the architecture. The ALE 1.0 specification provided only an interface for reading data (not writing), and only provided access to EPC data. The present ALE 1.1 specification expands upon ALE 1.0 to address writing as well as reading, and both the reading and writing aspects address not only EPC data but also other data that may be present on EPC data carriers. In particular, the ALE 1.1 specification is designed to provide full access to the functionality of the EPCglobal UHF Class 1 Gen 2 [Gen2] specification, when interacting with Gen2 RFID Tags. This includes reading and writing all memory banks, as well as exercising specific operations such as "lock" and "kill." In ALE 1.1, additional tag types may easily be accomodated in the future. In addition to providing reading and writing functionality, the ALE 1.1 specification also provides new interfaces for defining tag memory fields, for managing the naming of data source names ("logical readers"), and for securing the use of the APIs. A complete list of changes from the ALE 1.0 specification may be found in Section 16.

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 five means:

- It provides a means for clients to specify, in a high-level, declarative way, what data they are interested in or what operations they want performed, 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 or other devices 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 data and results from carrying out operations that is largely independent of where the data originated or how it was processed.
- It abstracts the channels through which data carriers are accessed 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 interact with 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.
- It abstracts the addressing of information stored on Tags and other data carriers into a higher-level notion of named, typed "fields," hiding from clients the details of how a particular data element is encoded into a bit-level representation and stored at a particular address within a data carrier's memory. This allows application logic to remain invariant despite differences between the memory organization of different data carriers (for example, differences between Gen 1 and Gen 2 RFID Tags), and also shields application logic from having to understand complex layout or data parsing rules.
- It provides a security mechanism so that administrators may choose which operations a given application may perform, as a policy that is decoupled from application logic itself.

The ALE specification does *not* specify a particular implementation strategy, or internal interfaces within a specific body of software. Instead, this specification focuses exclusively on external interfaces, admitting a wide variety of possible implementations so long as they fulfill the contract of the interfaces. For example, it is possible to envision an implementation of these interfaces as an independent piece of software that speaks to RFID readers using their network wire protocols. It is equally possible, however, to envision another implementation in which the software implementing these interfaces is part of the reader device itself.

The objectives of ALE as described above are motivated by twin architectural goals:

1. To drive as much filtering, counting, and other low-level processing as low in the architecture as possible (*i.e.*, in first preference to readers or other devices, then to low-level, application-independent software ("middleware" or embedded software), and as a last resort to "applications"), while meeting application and cost needs;
2. At the same time, to minimize the amount of "business logic" embedded in the tags, readers, embedded software/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, counting, writing, and other 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 more commonly it may be new software designed expressly to carry out an EPC-enabled operational process but which operates at a higher level than the "middleware" or embedded software 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.

In many cases, the client of ALE will be software that acts as an EPCIS Capturing Application (see Section 2) or other business processing software. Since EPCIS is another component of the EPCglobal Architecture Framework 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.

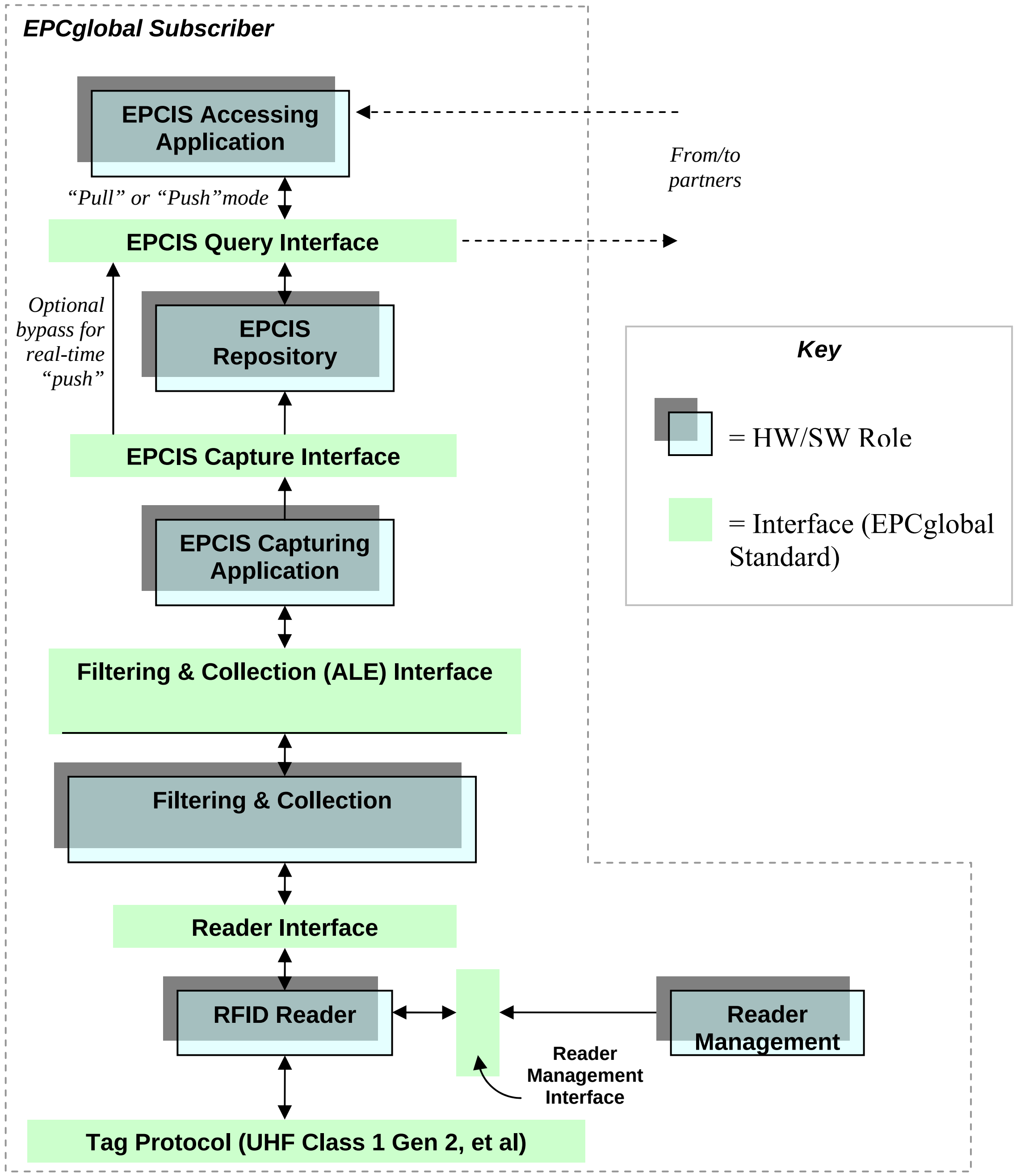
The distinction between the ALE and EPCIS/business layers is useful because it separates concerns. The ALE layer is concerned with dealing with the mechanics of data gathering, and of filtering down to meaningful events that are a suitable starting point for interpretation by business logic. Business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

For convenience, the ALE specification is divided into two parts. This Part I specifies at an abstract level all interfaces that are part of the ALE specification, using UML notation. Part II of the specification [ALE1.1Part2] specifies XML-based wire protocol bindings of the interfaces, including XSD schemas for all data types, WS-I compliant WSDL definitions of SOAP bindings of the service interfaces, and several XML-based bindings of callback interfaces used in certain modes of reading and writing data. Implementations may provide additional bindings of the API, including bindings to particular programming languages.

2 Role Within the EPCglobal Network Architecture

[Much of the text in this section is adapted from [EPCAF], Section 8.]

The diagram below shows the relationships between several EPCglobal Standards, from a data flow perspective. The plain green bars in the diagram below denote interfaces governed by EPCglobal standards, while the blue "shadowed" boxes denote roles played by hardware and software components of a typical system architecture. In any given deployment the mapping of roles in this diagram to actual hardware and software components may not be one-to-one. For example, in one deployment the "Filtering and Collection" role may be implemented by a software component and the "RFID Reader" role implemented by a hardware component. In another deployment, a "smart reader" may combine the "Filtering and Collection" role and the "RFID Reader" role into a single hardware component.



Consider a typical use case involving the reading of RFID Tags. Several processing steps are shown in the figure, each mediated by an EPCglobal standard interface. At each step progressing from raw tag reads at the bottom to EPCIS data at the top, the semantic content of the data is enriched. Following the data flow from the bottom of the figure to the top:

- *Readers* Make multiple observations of RFID tags while they are in the read zone.

- *Reader Interface* Defines the control and delivery of raw tag reads from Readers to the Filtering & Collection role. Events at this interface say "Reader A saw EPC X at time T."
- *Filtering & Collection* This role filters and collects raw tag reads, over time intervals delimited by events defined by the EPCIS Capturing Application (e.g. tripping a motion detector).
- *Filtering & Collection (ALE) Interface* Defines the control and delivery of filtered and collected tag read data from Filtering & Collection role to the EPCIS Capturing Application role. Events at this interface say "At Location L, between time T1 and T2, the following EPCs were observed," where the list of EPCs has no duplicates and has been filtered by criteria defined by the EPCIS Capturing Application.
- *EPCIS Capturing Application* Supervises the operation of the lower EPC elements, and provides business context by coordinating with other sources of information involved in executing a particular step of a business process. The EPCIS Capturing Application may, for example, coordinate a conveyor system with Filtering & Collection events, may check for exceptional conditions and take corrective action (e.g., diverting a bad case into a rework area), may present information to a human operator, and so on. The EPCIS Capturing Application understands the business process step or steps during which EPCIS data capture takes place. This role may be complex, involving the association of multiple Filtering & Collection events with one or more business events, as in the loading of a shipment. Or it may be straightforward, as in an inventory business process where there may be "smart shelves" deployed that generate periodic observations about objects that enter or leave the shelf. Here, the Filtering & Collection-level event and the EPCIS-level event may be so similar that no actual processing at the EPCIS Capturing Application level is necessary, and the EPCIS Capturing Application merely configures and routes events from the Filtering & Collection interface directly to an EPCIS Repository.
- *EPCIS Capture Interface* The interface through which EPCIS data is delivered to enterprise-level roles, including EPCIS Repositories, EPCIS Accessing Applications, and data exchange with partners. Events at this interface say, for example, "At location X, at time T, the following contained objects (cases) were verified as being aggregated to the following containing object (pallet)."
- *EPCIS Accessing Application* Responsible for carrying out overall enterprise business processes, such as warehouse management, shipping and receiving, historical throughput analysis, and so forth, aided by EPC-related data.
- *EPCIS Repository* Records EPCIS-level events generated by one or more EPCIS Capturing Applications, and makes them available for later query by EPCIS Accessing Applications.

The interfaces within this stack are designed to insulate the higher levels of the stack from unnecessary details of how the lower levels are implemented. One way to understand this is to consider what happens if certain changes are made:

- The Reader Interface insulates the higher layers from knowing what reader makes/models have been chosen. If a different reader is substituted, the information at the Reader Interface remains the same. The Reader Interface may, to some extent, also provide insulation from knowing what tag protocols are in use, though obviously not when one tag type or tag protocol provides fundamentally different functionality from another.
- The Filtering & Collection (ALE) Interface insulates the higher layers from the physical design choices made regarding how tags are sensed and accumulated, and how the time boundaries of events are triggered. If a single four-antenna reader is replaced by a constellation of five single-antenna "smart antenna" readers, the events at the Filtering & Collection level remain the same. Likewise, if a different triggering mechanism is used to mark the start and end of the time interval over which reads are accumulated, the Filtering & Collection event remains the same.
- The EPCIS interfaces insulate enterprise applications from understanding the details of how individual steps in a business process are carried out at a detailed level. For example, a typical EPCIS event is "At location X, at time T, the following cases were verified as being on the following pallet." In a conveyor-based business implementation, this likely corresponds to a single Filtering & Collection event, in which reads are accumulated during a time interval whose start and end is triggered by the case crossing electric eyes surrounding a reader mounted on the conveyor. But another implementation could involve three strong people who move around the cases and use hand-held readers to read the EPC codes. At the Filtering & Collection level, this looks very different (each triggering of the hand-held reader is likely a distinct Filtering & Collection event), and the processing done by the EPCIS Capturing Application is quite different (perhaps involving an interactive console that the people use to verify their work). But the EPCIS event is still the same.

In summary, the different steps in the data path correspond to different semantic levels, and serve to insulate different concerns from one another as data moves up from raw tag reads towards EPCIS.

The discussion above illustrated the relationships using a tag reading example, in which the flow of data was essentially one-way from the bottom of the diagram towards the top. Other scenarios, such as tag writing scenarios, may involve different directions of data flow, but the abstraction levels represented by the interfaces remain the same. For example, a manufacturing application may involve a step where a product bar code is read and an RFID tag written based on information read from the bar code. In that case, the "EPCIS Capturing Application" is responsible for coordinating the bar code read and the RFID tag write operations, each of which may involve a single event at the ALE level, which in turn correspond to multiple events at the lower levels. The essential role of ALE of insulating the capturing application from the physical details of how reads and writes are carried out remains the same.

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 Interfaces

The ALE specification defines five interfaces, as defined below.

Interface	Description	Normative Sections of This Document
Reading API	An interface through which clients may obtain filtered, consolidated EPC and other data from a variety of sources. In particular, clients may read RFID tags using RFID readers.	Sections 5, 6, 8
Writing API	An interface through which clients may cause operations to be performed on EPC data carriers through a variety of actuators. In particular, clients may write RFID tags using RFID "readers" (capable of writing tags) and printers.	Sections 5, 6, 9
Tag Memory Specification API	An interface through which clients may define symbolic names that refer to data fields of tags.	Section 7
Logical Reader Configuration API	An interface through which clients may define logical reader names for use with the Reading API and the Writing API, each of which maps to one or more sources/actuators provided by the implementation.	Section 10

Interface	Description	Normative Sections of This Document
Access Control API	An interface through which clients may define the access rights of other clients to use the facilities provided by the other APIs.	Section 11

Table 1. ALE APIs

To comply with this specification, an implementation of a given ALE API SHALL fully implement that API according to this specification. The specification permits a system component to include mutually related implementations of more than one ALE API. In the remainder of this document, the phrase "ALE implementation" refers to an implementation of one of the five ALE APIs, or to related implementations of two or more ALE APIs, the specific APIs involved being evident from the context in which the phrase is used. Accordingly, due to the manner in which the ALE 1.1 Specification is written, necessary implementations include any of the individual APIs and any and all combinations of APIs permitted under the ALE 1.1 Specification.

Explanation (non-normative): The ALE specification is designed to be applicable to a wide variety of implementations, including full-featured "middleware" software that controls multiple devices, as well as embedded implementations that provide only limited functionality. For example, there may be an implementation of the ALE Reading API embedded on a reader device that is only capable of reading tags, not writing them. Such an implementation has no need to provide the Writing API. Likewise, an implementation embedded on a single-antenna RFID reader is not likely to need the facilities of the Logical Reader Configuration API.

The remaining sections of this document specify these interfaces.

4.1 UML Notation for APIs

In each of sections noted in the table above, an API is described abstractly using UML class diagram notation. An implementation of an API is realized through one or more bindings of the UML to a specific implementation technology and message syntax.

The class notation used for the abstract UML specifications of classes and interfaces is illustrated below:

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

Data members are indicated above the dividing line ("---"), while interface methods are defined below the dividing line. In general, each API specifies an interface (marked with the `<<interface>>` stereotype) containing only methods. The methods in turn may use complex types as arguments or return types. Each complex type is specified by a class containing only data members.

Within the UML descriptions, the notation `<<extension point>>` identifies a place where implementations MAY add an extension, in compliance with this specification, through vendor specific additions of new data members or methods. The exact mechanism for extensibility is binding-specific. The extensibility mechanism provided by all bindings provides 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.

4.2 API Interaction

The general interaction model for each API is that there are one or more clients that make method calls to an interface class corresponding to an API, where the interface class is specified in one of the sections given in the table above. Each method call is a request, which causes the ALE implementation to take some action and return results. Thus, methods of each API are synchronous.

The Reading API and Writing API 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`. Formally, the path for asynchronous delivery is denoted by a "callback" interface. Different ALE implementations typically provide a variety of bindings of the Reading API or Writing API callback interface (*e.g.*, HTTP, file, e-mail, message bus, SOAP, *etc.*); this is intended to be a point of extensibility. Part II specifies bindings that are standardized, and specify the conformance requirement (MAY, SHOULD, SHALL) for each.

4.3 Version Introspection Methods

Each of the five APIs includes a pair of methods having the following signature:

```
---
getStandardVersion() : String
getVendorVersion() : String
```

An ALE implementation SHALL implement these methods as specified in the following table:

Method	Description
`getStandardVersion`	Returns a string that identifies what version of the specification this implementation of the API 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 API implementation fully complies, and SHOULD return the string corresponding to the latest version to which it complies. To indicate compliance with this Version 1.1 of the ALE specification, the implementation SHALL return the string `1.1`.
`getVendorVersion`	Returns a string that identifies what vendor extensions of the API 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 of the API 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 an Internet 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.

Table 2. Version Introspection Methods

Each of the five APIs defined in this specification includes a `getStandardVersion` and a `getVendorVersion` method. The result returned by each method SHALL only pertain to the API to which it belongs. For example, a system component might include an implementation of the Reading API that returns the string `1.0` from the `getStandardVersion` method, and an implementation of the Writing API that returns the string `1.1` from the `getStandardVersion` method. This would indicate that the system component's Reading API implementation complies with the ALE 1.0 specification but not the ALE 1.1 specification, while the Writing API implementation does comply with the ALE 1.1 specification.

4.4 Classes Common to the Reading and Writing APIs

The following seven classes are used by both the Reading API and the Writing API. While their names begin with the `EC` prefix used for Reading API classes, they should be understood as belonging equally to the Reading API and the Writing API.

Class	Specified in Section
`ECTime`	8.2.2
`ECTimeUnit`	8.2.3
`ECTrigger`	8.2.4
`ECFilterListMember`	8.2.8
`ECFieldSpec`	8.2.12
`ECReaderStat`	8.3.10
`ECSightingStat`	8.3.11

Table 3. Classes Common to the Reading and Writing APIs

4.5 Interpretation of Names

There are several places in the ALE APIs where an ALE client specifies a name to refer to an entity with which the API is concerned. For example, in the ALE Reading API, a client specifies an ECSpec name to refer to an event cycle specification (ECSpec). This section specifies treatment of names that applies to all places where names are used.

Except as noted elsewhere in this specification, an ALE implementation SHALL accept as a name any non-empty string of Unicode characters that does not include Pattern_White_Space or Pattern_Syntax characters (as those classes are defined in [Unicode]). An ALE implementation MAY accept other non-empty strings as well.

In many situations, a client provides a name to an ALE API in order to refer to an entity previously defined. This implies that an ALE implementation is called upon to recognize that a name specified by a client is the same as a name previously specified. For the purposes of the following sentence, these two names are referred to as the "specified name" and the "previously specified name," respectively. An ALE implementation SHALL consider the specified name equivalent to the previously specified name if it is an identical sequence of Unicode characters, MAY consider the specified name equivalent to the previously specified name if they are canonical-equivalent sequences (as the term "canonical-equivalent sequence" is defined in [Unicode]), and SHALL NOT consider the specified name equivalent to the previously specified name if they are not canonical equivalent sequences (except in situations of aliasing explicitly noted elsewhere in this specification).

In other situations, an ALE implementation returns a value to an ALE client that includes a name previously specified by an ALE client. In such situations, the name included in the returned value SHOULD be the identical sequence of Unicode characters as previously specified by the client, but MAY be a canonical-equivalent sequence.

Explanation (non-normative): The above rules are designed to give ALE clients a minimum set of reasonable behavior on which they can rely, without overly burdening implementations. The rules for construction of names give clients a wide range of strings that are guaranteed to be acceptable, while not requiring implementations to perform any checks (other than the test for non-empty). The rules for equality of names insure that identical strings will be treated as equal and that different strings will be treated as different. Implementations are permitted, but not required, to treat different yet canonical-equivalent sequences as equal; this means that implementations do not necessarily have to understand Unicode rules for combining marks. A consequence of these rules is that identifiers are treated as case-sensitive.

4.6 Scoping of Names

Names as discussed in Section 4.5 exist within a namespace; the names within one namespace are unrelated to the names in other namespaces. An ALE implementation SHALL permit the same string to be used as a name in more than one namespace. The following table enumerates all namespaces that are implied by the ALE APIs. In the table below, the “global” scope refers to the ALE implementation as viewed by any one client; it is implementation-defined whether or not global namespaces are shared among different clients.

Explanation (non-normative): The last sentence allows different implementations to take different positions as to whether different users share data or not.

Namespace	Section	Scope
Fieldname	6, 7	Global
TMSpec name	7	Global
ECSpec name	8	Global
ECReport name	8.2.5	Enclosing ECSpec
CCSpec name	9	Global
CCCmdSpec name	9.3.2	Enclosing CCSpec
EPC Cache name	9.5	Global
Association Table name	9.6	Global
Random Number Generator name	9.7	Global
Logical Reader name	10	Global
Permission name	11	Global
Role name	11	Global
Client Identity name	11	Global

4.7 Equivalance of Null, Omitted, and Empty String Values, and of Omitted and Empty Lists

Throughout this specification, data members may be noted as "optional" or "conditional." This means that the data member may lack a value in certain circumstances. Each binding of the ALE APIs provides its own representation for this situation. In some situations, more than one representation may be available. For example, in the XML bindings specified in [ALE1.1Part2], an optional data member may be omitted altogether, or may appear as an XML element or attribute having the empty string as its text content.

Within this specification, the terms "null," "omitted," and "empty string" are used interchangeably to denote an absent value. An implementation SHALL NOT draw any distinction between "null," "omitted," and "empty string." If a binding provides more than one representation as illustrated above, the ALE implementation SHALL treat them as equivalent.

Similarly, an implementation SHALL NOT draw any distinction between an omitted list and a list containing zero elements. If a binding provides more than one representation for this situation, the ALE implementation SHALL treat them as equivalent.

5 ALE Concepts and Principles of Operation

This section describes the concepts and principles of operation that underlie the specification of the ALE Reading API and the ALE Writing API.

5.1 Fundamental ALE Concepts

The purpose of the ALE interface is to allow business applications to read and operate upon tags. While ALE was primarily conceived and developed in the context of RFID tags, the interface is designed to be general enough to accommodate other kinds of data carriers, such as bar codes, OCR text, and in some instances even human interaction through a keyboard or display. Within this specification, the term "Tag" refers to a data carrier of this kind; that is, to an RFID tag or some other data carrier that can be treated in a similar manner.

Within this specification, the term "Reader" is used to refer to a channel through which Tags are accessed. Through a Reader, data may be read from Tags, and in some cases (depending on the capabilities of the Readers and Tags involved) data may be written to Tags or other operations performed on Tags. An extremely common type of Reader, of course, is an actual RFID reader, which accesses RFID tags through an RFID air interface. But a Reader could just as easily be a bar code reader or even a person typing on a keyboard (in both of those examples, data may be read but not written). 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 10. Hence, the term "Reader" is just a convenient shorthand for "channel for accessing Tags." 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 access 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 *reader cycle* is the smallest unit of interaction with a Reader. As ALE permits a wide variety of Readers, the exact nature of a reader cycle is highly dependent on the particular kind of channel a given Reader represents. For example, for an ALE implementation embedded in an RFID reader device, the "Reader" is the communication pathway between the ALE subsystem and the RF protocol subsystem, and a reader cycle might represent one iteration of the RF protocol used to communicate with RFID tags. Another example is an ALE implementation provided by middleware software, which communicates with an outboard RFID reader device through a proprietary wire protocol. In this case, a reader cycle is a unit of interaction defined by that wire protocol, which may correspond to one or several RF protocol iterations depending on the design of the reader device and its wire protocol.

An *event cycle* or *command cycle* is an interval of time over which an ALE implementation carries out interactions with one or more Readers on behalf of an ALE client. (“Event cycle” is the term used in the reading API, while “command cycle” is the term used in the writing API.) It is the smallest unit of interaction between an ALE client and an ALE implementation. An ALE client describes declaratively what it wants to accomplish during one or more event cycles, for example “read from readers A and B for five seconds, and report any tags whose EPCs match the product code for Acme Widgets.” The event cycle is the five second interval during which the ALE implementation carries out the client's request (in this example, reading tags from readers A and B and filtering as specified).

A *report* is a response sent from the ALE implementation to the ALE client at the conclusion of an event cycle or command cycle. The report contains information about what happened during the event cycle or command cycle: information read from Tags (for an event cycle) or confirmation of Tags written or otherwise manipulated (for a command cycle). A report is typically sent to the ALE client at the end of each event cycle or command cycle, although the ALE client may ask that reports be suppressed if nothing “interesting” occurred during the event cycle or command cycle (e.g., if no tags were read).

In general, during an event cycle or command cycle an ALE implementation carries out one or more reader cycles with the designated Readers, and through those reader cycles carry out the wishes of the ALE client for that event cycle or command cycle. This specification, however, does not stipulate how an ALE implementation must employ reader cycles to fulfill ALE client requests. The ALE implementation has wide latitude to interact with Readers in whatever manner it determines is appropriate, so long as the net effect as seen by the ALE client conforms to this specification. Likewise, this specification makes no assumption about the granularity of reader cycles in terms of how much work is performed in a single reader cycle. With this approach, the ALE specification recognizes that different kinds of Readers may operate differently, with wide differences in what a reader cycle is, and hence an ALE implementation may have to employ different strategies depending on these characteristics.

For these reasons, the ALE APIs specified herein do not expose reader cycles to clients in any direct way. The APIs are specified by defining event cycles and command cycles as seen by ALE clients. The only reason that the term reader cycle has been introduced is to aid in explaining certain aspects of event cycles and command cycles that would be difficult to explain otherwise.

5.2 Event Cycles

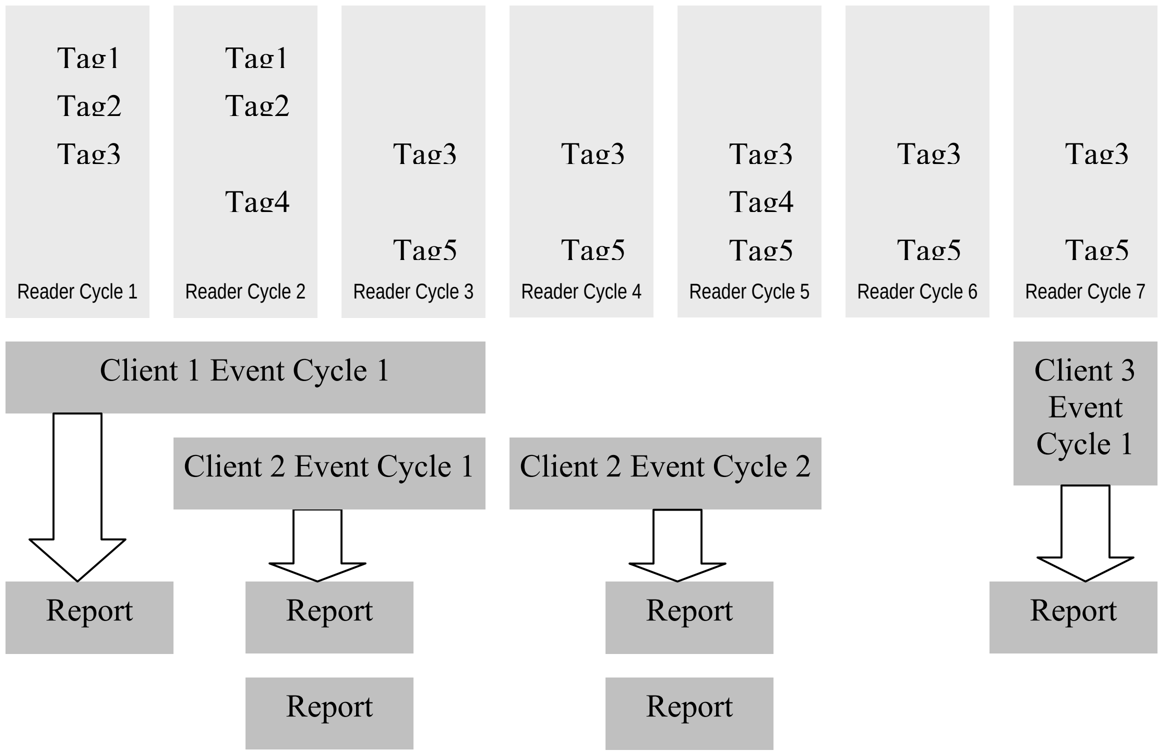
An event cycle is the smallest unit of interaction between an ALE client and an ALE implementation through the ALE Reading API. An event cycle is an interval of time during which Tags are read. At the conclusion of an event cycle, a report is sent to the ALE client containing information read from the Tags.

As Tags move in and out of the detection zone of a Reader, the tag data reported to the ALE implementation by the Reader changes. 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). An ALE client specifies when an event cycle starts and stops. An ALE client may specify that an event cycle may:

- Extend for a specified duration (interval of real time); *e.g.*, accumulate reads into five-second intervals.
- Occur periodically; *e.g.*, read for one minute once every 30 minutes.
- 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 Tags are detected by any Reader specified for that event cycle for a specified interval of time.
- Terminate when any Reader specified for that event cycle reports a new Tag to the ALE implementation, thus delivering data to the ALE client as soon as it is known to the ALE implementation..

The complete set of available options is specified normatively in Section 8.2.1.

The net picture looks something like this:



While the diagram shows reader cycles arising from a single Reader, in practice a given event cycle may collect reader 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 reader 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 tag data are shared by all active event cycles that request data from a given Reader.

The set of Tags in a given reader cycle from a given Reader is denoted by *S*. In the picture above, *S1* = {Tag1, Tag2, Tag3} and *S2* = {Tag1, Tag2, Tag4}. Each Tag, in turn, is modeled as a tuple of data fields: Tag1 = (Tag1Field1, Tag1Field2, …).

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 Tags read in any of the reader cycles that take place during 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 set of Tags from each reader cycle: $E = S1 \cup S2 \cup \ldots$. In the example above for Client 1 Event Cycle 1 we have *E1.1* = {Tag1, Tag2, Tag3, Tag4, Tag5}.

ALE 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, which includes some Tags and excludes others based on the data contained in their fields.

- Whether to report

    - The members of the set, *F(R)* (*i.e.*, the tag data themselves), possibly grouped as described in Section 5.2.1. In this case, the ALE client also specifies which data fields to report for each Tag, and how the data is to be formatted for consumption by the client;

    - The quantity, or cardinality, of the set |*F(R)*|, or of the groups making up the set as described in Section 5.2.1.

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 is as follows:

1. A client provides to the ALE implementation an *event cycle specification* (ECSpec), which specifies

    - one or more Readers (this is done indirectly, as explained in Section 10)

    - event cycle boundaries as illustrated above, and

    - a set of reports as defined above

2. The ALE Layer responds by returning the information implied by that report specification for one or more event cycles.

This interaction may take place in a "pull" mode, where the client provides the ECSpec 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 to an ECSpec, 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.

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.

A key characteristic of the ALE Reading API is that Tags may be read several times within an event cycle. This is necessary so that data may be shared between simultaneous event cycles that share the same Readers but specify different time boundaries. A fundamental characteristic of event cycles is that duplicate reads are removed before the data is presented to the ALE client. The formal model expresses this by defining the Tags S as a *set*, as opposed to a list. This implies the existence of an algorithm for determining whether two Tags reported in successive reader cycles are the "same" for the purposes of duplicate removal. Likewise, the additions and deletions options for reporting rely on the set difference operator, which also requires comparison of tag data. This topic is addressed more fully in Section 8.2.

5.2.1 Group Reports

Sometimes it is useful to group Tags read during an event cycle based on portions of the EPC or other fields. For example, in a shipment receipt verification application using SGTIN EPCs, 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 tag data into some sort of group code g. For example, a grouping operator might map the EPC field of a tag 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 a tag, such as the filter code that implies the type of object (*i.e.*, pallet, case, item, *etc.*), a lot code in a field of user memory, and so on.

The notation $S{\downarrow}g$ means the subset of Tags $s1$, $s2$, … 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: $\{\, (g, F(R){\downarrow}g) \mid F(R){\downarrow}g \text{ is non-empty} \,\}$. In this case, the ALE client also specifies which data fields to report for each Tag, and how the data is to be formatted for consumption by the client.
  - 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

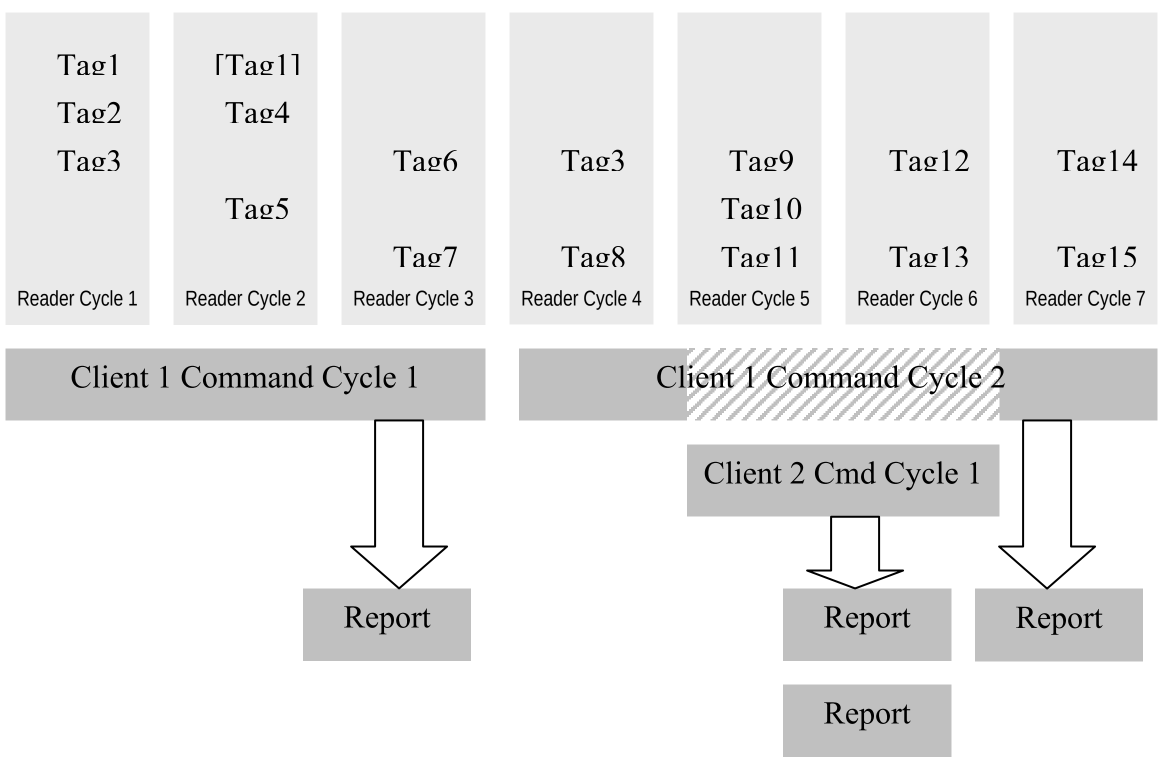
(unnamed) group. In mathematical notation: $\{\, (g, |F(R)\!\downarrow\! g|) \mid F(R)\!\downarrow\! g$ is non-empty $\}$.

5.3 Command Cycles

A command cycle is the smallest unit of interaction between an ALE client and an ALE implementation through the ALE Writing API. A command cycle is an interval of time during which Tags are written, or other operations performed upon them (e.g., the "kill" and "lock" operations available for UHF Class 1 Gen 2 RFID tags). At the conclusion of a command cycle, a report is sent to the ALE client containing information about what tags were operated upon and what the results were.

As in an event cycle, the ALE client specifies when a command cycle starts and stops. During the command cycle, the ALE implementation uses one or more Readers to operate upon Tags that fall within the detection zone of the Readers. The implementation makes best efforts to acquire and operate on each Tag exactly once.

The net picture looks something like this:



While the diagram shows command cycles each using a single Reader, in practice a given command cycle may use more than one Reader to acquire tags.

The interaction between an ALE client and an ALE implementation through the Writing API is similar to the description of the Reading API from the last section. Namely,

1. A client provides to the ALE implementation a *command cycle specification* (CCSpec), which specifies

- one or more Readers (this is done indirectly, as explained in Section 10)
- command cycle boundaries, and
- a set of command lists to apply to Tags. Each command list includes
    - a filter that specifies which Tags to operate upon, and
    - an ordered list of operations to perform on each Tag that matches the filter.

2. The ALE Layer responds by carrying out the operations on Tags, and returning a report that describes what Tags were encountered and what processing was performed upon them.

As in the Reading API, this interaction may take place in a "pull" mode, where the client provides the CCSpec and the ALE Layer in turn carries out Tag operations and returns the report(s). It may also be done in a "push" mode, where the client registers a subscription to a CCSpec, and thereafter the ALE Layer asynchronously sends reports to the client when command cycles complete.

A key difference between event cycles and command cycles is the way that simultaneous use of the same Readers is treated, and the implications upon the way implementations are expected to acquire Tags. Event cycles only read Tags, without changing their contents or performing other side-effects upon them. Hence, it is possible for several simultaneously active event cycles to share the result of a single reader cycle, and an ALE implementation MAY share reader cycles in this way. Because simultaneous event cycles may have different boundaries, it MAY be necessary for the ALE implementation to read a given Tag more than once. Duplicate detection is done on the basis of data read from the Tag; an event cycle specification indicates which fields of Tag data are to be used for this purpose.

In contrast, command cycles may write Tags and perform other side-effects such as killing or locking. Because a command cycle changes the data on a Tag, using Tag data itself may not be a reliable method to determine duplicates. Instead, an ALE implementation SHOULD use other means to singulate tags within a command cycle. For example, Gen2 RFID Tags have inventory flags which can be used. Simultaneous command cycles are permitted in the ALE Writing API, but it is not expected that reader cycles will be shared. This is both because simultaneous command cycles are likely to be operating upon disjoint sets of Tags or performing disjoint operations on them, and because each command cycle may need to do its own bookkeeping to avoid duplicates (e.g., in Gen2 two simultaneous command cycles could use different sessions for singulation). Therefore, while any command cycles are active it is expected that each Reader will be dedicated to a single command cycle (or the set of all event cycles) during any given reader cycle. The ALE implementation MAY apply whatever rules it wishes to determine which command or event cycles get access to a Reader during any reader cycle. In the illustration above, for example, Client 2 Command Cycle 1 has pre-empted the use of the Reader by Client 1 Command Cycle 2 for the entire duration of the former.

5.4 Tag Data Model

From the perspective of an ALE client, the data on a Tag is considered to consist of one or more data fields. When an ALE client describes an event cycle or a command cycle, for each data field it wants to operate upon the client must specify a *fieldspec*. A fieldspec specifies three things:

- A *fieldname*, which specifies which data field of the Tag to operate upon.
- The *datatype*, which specifies what kind of data values that field is considered to contain, and how they are encoded into the Tag memory
- A *format*, which specifies the syntax by which individual data values are presented at the level of the ALE API (that is, the format of data values as reported by the ALE API when fields are read, and the format of data values provided by the ALE client to the ALE API as input to a write operation or a filtering specification).

The following table gives examples to illustrate these concepts:

Fieldname	Datatype	Format
Bits 0-15 of the User Memory bank (bank 11)	Integer, encoded in two's complement binary with the least significant bit in bit 15	Decimal numeral, with no leading zeros and an optional minus sign. Alternately, a hexadecimal numeral.
The EPC bank of a Gen2 tag, according to Section 3.2 of the EPC Tag Data Standards	An EPC, encoded according to Section 3 of the EPC Tag Data Standards	A tag URI as defined in Section 4 of the EPC Tag Data Standards. Alternately, a raw Hex URI as defined in Section 4.3.9 of the EPC Tag Data Standards
The field with OID 12345 in user memory of a Gen2 tag that is encoded according to ISO 15962	A timestamp, encoded as seconds since Midnight GMT January 1, 1970.	An ISO-8601 compliant string of the form yyyy-mm-ddThh:mm:ss[TZ]

Table 4. Illustration of Fieldname, Datatype, and Format

As the above table suggests, there might be more than one format that is usable with a given data type.

The ALE API is intended to provide high-level access to memory fields in a way that shields ALE clients from being concerned with low-level details of how memory fields are arranged and compacted into Tag memory. In general, there are two broad classes of fields. A field that occupies a fixed location in Tag memory is a *fixed field*. A field that does not occupy a fixed location or that may be absent is a *variable field*.

The ALE API provides three kinds of fieldnames:

- Fixed-address fieldnames of the form `@`*`bank`*`.`*`length`*`[.`*`offset`*`]`, where *`bank`*, *`length`*, and *`offset`* are integers. A fieldname of this form specifies a fixed field comprising *`length`* contiguous bits, starting at fixed bit location *`offset`* within bank *`bank`* of tag memory. Fieldnames of this form are specified in detail in Section 6.1.9.1.
- Variable fieldnames of the form `@`*`bank`*`.`*`oid`*, where *`bank`* is an integer and *`oid`* is an object identifier expressed as a URN according to [RFC3061]. A fieldname of this form specifies a variable field encoded according to ISO 15962 [ISO15962]. Fieldnames of this form are specified in detail in Section 6.1.9.2.
- A symbolic fieldname that is a user- or implementation-defined string, not beginning with an atsign (`@`) character. Within this category, there are four variants:
  - A symbolic fieldname may be one of the standardized names defined in Section 6.1.
  - A symbolic fieldname may be defined by the ALE client using the `TMFixedFieldListSpec` of the Tag Memory API (Section 6.2.3). This form allows an ALE client to define one or more symbolic names that are equivalent to fixed field fieldnames of the form `@`*`bank`*`.`*`length`*`[.`*`offset`*`]`. By assigning symbolic fieldnames that are meaningful to an application, applications that define event cycles and command cycles through the ALE API are insulated from knowing exactly how fields are laid out in tag memory.
  - A symbolic fieldname may be defined by the ALE client using the `TMVariableFieldListSpec` of the Tag Memory API (Section 7). This form allows an ALE client to define one or more symbolic names that are equivalent to variable fieldnames of the form `@`*`bank`*`.`*`oid`*.
  - A symbolic fieldname may be defined by the ALE client using a vendor extension to the Tag Memory API (Section 7). Through vendor extension, ALE implementations MAY provide more sophisticated ways of mapping symbolic names to tag memory. In particular, ALE implementations MAY provide for a single symbolic name to map to a different address in tag memory depending on the type of Tag being accessed. An ALE implementation MAY also provide for mapping schemes that are either fixed or variable.

5.4.1 Default Datatype and Format

A given fieldname always implies a default datatype and a default format. In the ALE Reading API and the ALE Writing API, an ALE client may refer to a fieldname without explicitly providing a datatype or format, in which case the default datatype and format are used. The ALE client may, however, supply an explicit datatype or format that overrides the default. For a fieldname of the form `@`*`bank`*`.`*`length`*`[.`*`offset`*`]`, the default datatype is unsigned integer and the default format is hexadecimal as defined in Section 6.2.2. For a fieldname of the form `@`*`bank`*`.`*`oid`*, the default datatype is `iso-15962-string` and the default format is string as defined in Section 6.2.3. For a symbolic fieldname, the default datatype and format are specified when the symbolic name is defined.

5.4.2 “Field Not Found” Condition

When an ALE implementation accesses a particular Tag during an event cycle or command cycle, it may be that the Tag does not have a field that is specified in the governing ECSpec or CCSpec. This is called a “field not found” condition. A “field not found” condition can arise for several reasons. For example:

- A field is defined to be at a fixed offset within a specific bank of Tag memory, but the Tag being accessed does not have that bank or the size of the bank does not extend as far as the field’s offset.
- A field is only defined for certain Tag types. For example, a field in the “user memory” bank, which is not defined for a Gen1 tag.
- A field is defined to be at a variable position in a scheme that encodes fields as a series of name/value pairs, and the specified field does not exist among the fields encoded in Tag memory.
- A field is defined to be at a variable position using a directory-based scheme, and the tag’s directory lacks an entry for the specified field.

The definition of each fieldname specifies the conditions under which a “field not found” condition occurs.

A fixed field, as defined above, is defined to occupy a fixed location within Tag memory. Therefore, a fixed field always exists as long as the memory bank exists and is of sufficient size. A variable field, on the other hand, may or may not exist depending on the contents of memory. In addition to “read” and “write” operations, the ALE Writing API also supports “add” and “delete” operations on variable fields. A “delete” operation on a variable field will cause subsequent operations to result in a “field not found” condition.

In addition, variable fields may require certain information to be present in Tag memory locations other than the field itself. For example, a directory-based encoding scheme may require a directory to be present before any fields may be accessed. An “initialize” operation is provided to put the Tag memory into a state where such fields may be added, read, written, or otherwise operated upon. For fields that require initialization, an attempt to access such fields if the Tag memory has not been properly initialized will result in a “field not found” condition.

A “field not found” condition results in the following behavior in the Reading API:

- If the field was included in the `primaryKeyFields` list, it causes the Tag to be omitted from the event cycle.
- If the field was included in an `ECFilterSpec`, it is treated as a failure to match the pattern; that is, it causes the Tag to fail an INCLUDE filter or pass an EXCLUDE filter.

- If the field was included in an `ECGroupSpec`, it causes the Tag to be assigned to the default group.
- If the field was included in an `ECReportOutputSpec`, it causes the value to be reported as null.

A "field not found" condition results in the following behavior in the Writing API:

- If the field was included in an `ECFilterSpec`, it is treated as a failure to match the pattern; that is, it causes the Tag to fail an INCLUDE filter or pass an EXCLUDE filter.
- If the field was included in a `CCOpSpec`, it causes the operation to be reported with a `FIELD_NOT_FOUND_ERROR` status code.

5.4.3 "Operation Not Possible" Condition

When an ALE implementation accesses a particular Tag during an event cycle or command cycle, it may be that the Tag has a field that is specified in the governing ECSpec or CCSpec, but that the Tag does not support performing the requested operation on that field. This is called an "operation not possible" condition. For example, Gen2 RFID Tags only support the locking of an entire bank of memory, so an attempt to lock a field that maps to just a subset of a memory bank will result in an "operation not possible" condition. The definition of each fieldname specifies the conditions under which an "operation not possible" condition occurs. For the purposes of defining such conditions, a "read operation" refers not only to an explicit read operation in a CCSpec, but also the use of a fieldname in any ECSpec or CCSpec context that requires reading the contents of the field. This includes use of the fieldname in the `primaryKeyFields` parameter of an ECSpec (Section 8.2), in an `ECFilterSpec` (Section 8.2.7), in an `ECGroupSpec` (Section 8.2.9), and in an `ECReportOutputSpec` (Section 8.2.10).

An "operation not possible" condition results in the following behavior in the Reading API:

- If the field was included in the `primaryKeyFields` list, it causes the Tag to be omitted from the event cycle.
- If the field was included in an `ECFilterSpec`, it is treated as a failure to match the pattern; that is, it causes the Tag to fail an INCLUDE filter or pass an EXCLUDE filter.
- If the field was included in an `ECGroupSpec`, it causes the Tag to be assigned to the default group.
- If the field was included in an `ECReportOutputSpec`, it causes the value to be reported as null.

An "operation not possible" condition results in the following behavior in the Writing API:

- If the field was included in an `ECFilterSpec`, it is treated as a failure to match the pattern; that is, it causes the Tag to fail an INCLUDE filter or pass an EXCLUDE filter.
- If the field was included in a `CCOpSpec`, it causes the operation to be reported with a `OP_NOT_POSSIBLE_ERROR` status code.

5.4.4 "Out of Range" Condition

When an ALE implementation writes data to a particular tag during a command cycle, it may be that the value to be written is a legal value for its datatype, but cannot be encoded into the specified field. This is called an "out of range" condition. For example, any nonnegative integer is legal as a value for the `uint` datatype, but only numbers less than 256 can be encoded in a fixed field of eight bits. An attempt to write the number 500 into an 8-bit fixed field would raise an "out of range" condition.

If execution of a `CCOpSpec` results in an "out of range" condition, the operation is reported with an `OUT_OF_RANGE_ERROR` status code. Unlike the "field not found" and "operation not possible" conditions, an "out of range" condition cannot occur merely because a field is included in an `ECFilterSpec`, nor in any part of the Reading API.

5.4.5 Pattern Fieldnames

The ALE Reading API permits the client to specify a specific field to be read, or to specify that a set of related fields are to be read. The latter is specified by the use of a "pattern fieldname." When a pattern fieldname is used in a fieldspec, the datatype and format must be valid for all fields that match the pattern. Pattern fieldnames may only be used in a fieldspec that occurs as part of a `ECReportOutputFieldSpec` (Section 8.2.11)

5.5 Reader Cycle Timing

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

- A client or clients may make simultaneous requests for event cycles 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 reader cycles take place. Specifying the reader cycle frequency outside of any event cycle request insures that clients cannot make contradictory demands on reader cycles.
- In cases where there are many RFID readers in physical proximity (perhaps communicating to different ALE implementations), the reader 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 reader cycle timing, it would necessarily make assumptions about the source of reader 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 reader 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 reader cycle timing. Implementations 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 reader cycle timing, the ALE implementation always has the freedom to suspend Reader activity during periods when no event cycles or command cycles using a given Reader are active.

5.6 Execution of Event Cycles and Command Cycles

An event cycle specification (ECSpecs) or a command cycle specification (CCSpecs) comes into existence through a client interacting with the ALE Reading API or the ALE Writing API, respectively. Once created, an ECspec or CCSpec (hereafter abbreviated to EC/CCSpec) is subject to a lifecycle that is governed by subsequent client interactions through the Reading/Writing API as well as events related to the boundary conditions specified as part of the EC/CCSpec. Event/command cycles occur, and reports are generated, within the lifecycle of an EC/CCSpec as specified below.

Normative specifications of the ALE Reading API and the ALE Writing API are found in Sections 8 and 9, respectively. The following is an informal description, to help provide context for the EC/CCSpec lifecycle state diagrams specified below.

In both the ALE Reading API and the ALE Writing API, there are two ways to create an event/command cycle. A standing EC/CCSpec may be posted using the `define` method of the Reading/Writing API. Subsequently, one or more clients may subscribe to that EC/CCSpec using the `subscribe` method. The EC/CCSpec will execute event/command cycles as long as there is at least one subscriber. A `poll` call is like subscribing then unsubscribing immediately after one event/command cycle is completed (except that the results are returned from `poll` instead of being sent to a subscriber asynchronously). The second way to create an EC/CC spec is to post it for immediate execution using the `immediate` method. This is roughly equivalent to defining an EC/CCSpec, performing a single `poll` operation, and then undefining it.

The lifecycle of EC/CCSpecs is defined with the aid of a state diagram having three states:

State	Description (informal)

State	Description (informal)
Unrequested	The EC/CCSpec has been defined, but no client has expressed interest by subscribing or polling.
Requested	The EC/CCSpec has at least one client that is interested, but Tags are not currently being processed for an event/command cycle.
Active	Tags are currently being processed for an event/command cycle.

Table 5. EC/CCSpec Lifecycle States

By definition, an EC/CCSpec created by the `immediate` method cannot be in the *unrequested* state. Standing EC/CCSpecs that are requested using `subscribe` may transition in and out the *active* state multiple times. EC/CCSpecs that are requested using `poll` or created using `immediate` will transition in and out of the *active* state just once (though in the case of `poll`, the EC/CCSpec remains defined afterward so that it could be subsequently polled again or `subscribe`d to).

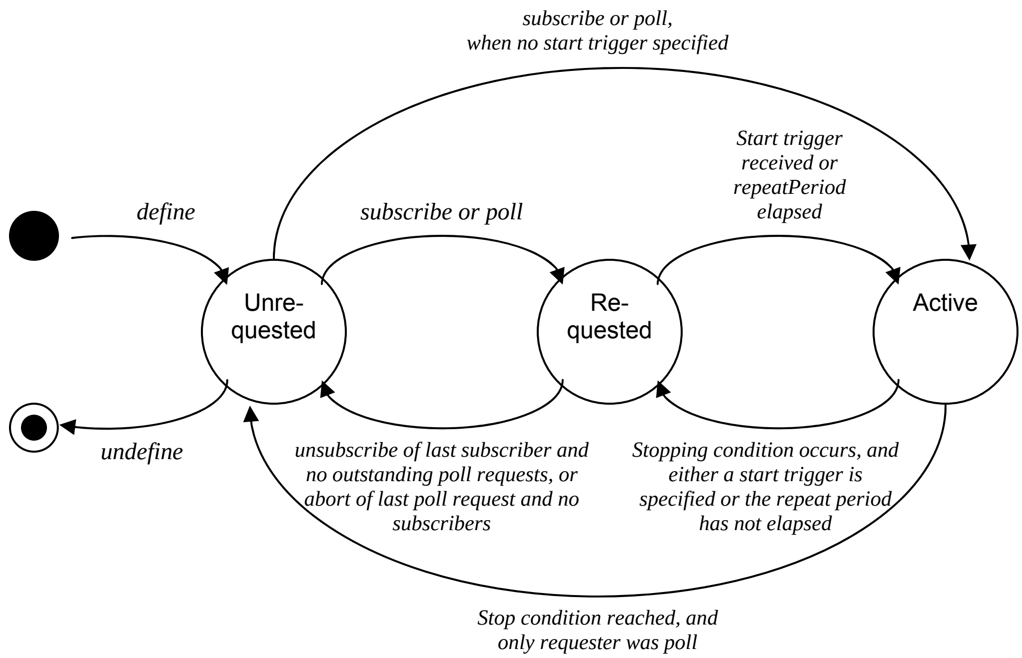
The complete normative specification of the state transitions is specified in Sections 5.6.1 and 5.6.2, below.

5.6.1 Lifecycle State Transitions for EC/CCSpecs Created by the Define Method

An EC/CCSpec that is created by a call to the `define` method of the ALE Reading/Writing API SHALL begin in the *unrequested* state, with an empty set of subscribers. Thereafter, it is subject to state transitions that occur in response to the following kinds of events:

- Calls to the `subscribe`, `unsubscribe`, `poll`, or `undefine` methods whose `specName` parameter refers to that EC/CCSpec.
- An outstanding `poll` call being aborted, as provided for in Sections 8.1 and 9.1.
- Event/command cycle starting and stopping conditions, as specified by the EC/CCSpec. The EC/CCSpec parameters that determine starting and stopping conditions are defined in Sections 8.2.1 and 9.3.1.

The principal state transitions are illustrated in the diagram below. For clarity, not all state transitions are shown in the diagram; the tables following the diagram constitute the normative specification of all state transitions.



An EC/CCSpec that is created by a call to the `define` method SHALL be subject to the state transitions specified in the three tables below. In these tables, "start triggers" and "repeat period" refer to start condition information that is derived from the EC/CCSpec as described normatively in Sections 8.2.1 and 9.3.1. It is possible for an EC/CCSpec to specify no start triggers or to specify no repeat period (though at least one stop condition must be specified), and this figures into the description of the state transitions.

The following transitions SHALL apply when the EC/CCSpec is in the *unrequested* state:

Event (when in the *unrequested* state)	**Action**	**Next state**
Call to `subscribe`	The specified subscriber is added to the set of current subscribers for the EC/CCSpec.	*Active*, if the EC/CCSpec does not specify any start triggers; *requested* otherwise
Call to `poll`	A new `poll` call is outstanding.	*Active*, if the EC/CCSpec does not specify any start triggers; *requested* otherwise
Call to `undefine`	All information associated with the EC/CCSpec, including the set of current subscribers, is discarded.	(EC/CCSpec no longer exists)

Table 6. State Transitions from the Unrequested State

The following transitions SHALL apply when the EC/CCSpec is in the *requested* state:

Event (when in the *requested* state)	Action	Next state
Call to `subscribe`	The specified subscriber is added to the set of current subscribers for the EC/CCSpec.	*Requested*
Call to `poll`	A new `poll` call is outstanding.	*Requested*
Call to `unsubscribe`	The specified subscriber is removed from the set of current subscribers for the EC/CCSpec.	*Unrequested*, if there are no more subscribers or outstanding `poll` calls; *requested* otherwise
An outstanding `poll` call is aborted by the ALE client	The call to `poll` is no longer outstanding.	*Unrequested*, if there are no more subscribers or outstanding `poll` calls; *requested* otherwise
Call to `undefine`	For each outstanding `poll` call that is a requester of this EC/CCSpec, a report is returned having `initiationCondition` set to `UNDEFINE`, `terminationCondition` set to `UNDEFINE`, and no content apart from the header. No reports are delivered to subscribers. All information associated with the EC/CCSpec, including the set of subscribers, is discarded.	(ECSpec no longer exists)
Arrival of a start trigger	An event/command cycle begins.	*Active*

Event (when in the ***requested*** state)	Action	Next state
The repeat period has elapsed since the most recent transition to the *active* state (from any other state), provided there have been no intervening transitions to the *unrequested* state.	An event/command cycle begins.	*Active*

Table 7. State Transitions from the Requested State

The following transitions SHALL apply when the EC/CCSpec is in the *active* state:

Event (when in the ***active*** state)	Action	Next state
Call to `subscribe`	The specified subscriber is added to the set of current subscribers for the EC/CCSpec.	*Active*
Call to `poll`	A new `poll` call is outstanding.	*Active*
Call to `unsubscribe`	The specified subscriber is removed from the list of current subscribers. The event/command cycle ends with no reports delivered, if there are no more subscribers or outstanding `poll` calls.	*Unrequested*, if there are no more subscribers or outstanding `poll` calls; *Active* otherwise
An outstanding `poll` call is aborted by the ALE client	The poll call is no longer outstanding. The event/command cycle ends with no reports delivered, if there are no more subscribers or outstanding `poll` calls.	*Unrequested*, if there are no more subscribers or outstanding `poll` calls; *Active* otherwise

Call to `undefine`	The event/command cycle ends. Reports are returned to all outstanding `poll` calls for this EC/CCSpec (and thereafter, those poll calls are no longer outstanding). Reports are delivered to all current subscribers, unless suppressed according to Sections 8.2.5 and 9.3.2. All reports SHALL have `terminationCondition` set to `UNDEFINE`. For an ECSpec, the reports SHALL include any Tags that were read prior to the `undefine` call. For a CCSpec, the reports SHALL include any operations that were completed prior to the `undefine` call. All information associated with the EC/CCSpec, including subscribers and prior tag set state, is discarded.	(ECSpec no longer exists)
A stopping condition has occurred, as specified in Section 8.2.1 (for an ECSpec) or Section 9.3.1 (for a CCSpec)	The event/command cycle ends. Reports are returned to all outstanding `poll` calls for this EC/CCSpec (and thereafter, those poll calls are no longer outstanding). Reports are delivered to all current subscribers, unless suppressed according to Sections 8.2.5 and 9.3.2.	*Active*, if a repeat period is specified and the repeat period has elapsed since the transition into the *active* state, or if neither a repeat period nor any start triggers are specified (either of these counts as a new transition into the *active* state for the purpose of describing transition events); *Requested*, otherwise

Table 8. State Transitions from the Active State

Events occuring at times other than those specified in the tables above SHALL NOT cause a state transition.

Explanation (non-normative): In general, subscribers receive reports when event or command cycles complete (that is, transition out of the active state*). Nothing is sent to* *subscribers to indicate that a subscription has been removed via* `unsubscribe`*, or that an ECSpec or CCSpec was removed via* `undefine` *(except in the case that an* `undefine` *causes a transition out of the* active *state).*

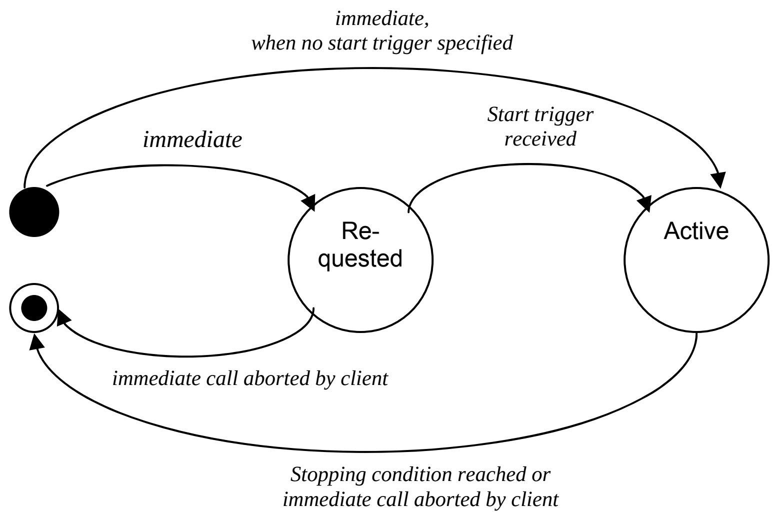
Special treatment is given in the Writing API to two or more simultaneous `poll` calls for the same CCSpec when parameters are supplied. Normally, simultaneous `poll` calls for the same CCSpec share the same command cycle, and results are delivered to all such `poll` calls when the command cycle completes. (The same is true for event cycles in the Reading API.) This cannot be done, however, if the CCSpec includes `CCOpSpec` instances that refer to parameters, and the simultaneous `poll` calls each supply different parameter values. If an ALE Writing API implementation receives a second `poll` call for a CCSpec for which there is already an outstanding `poll` call, and the second `poll` call specifies different parameter values, the ALE implementation SHALL satisfy the second `poll` by a initiating a new command cycle rather than sharing the results of the first, as though the second `poll` were of a different CCSpec. Because both command cycles share the same logical readers the two command cycles may fall subject to pre-emption as specified in Section 5.3. If an ALE implementation receives a second `poll` call for a CCSpec for which there is already an outstanding `poll` call, and the second `poll` call specifies the same parameter values as the first, the ALE implementation MAY treat the second `poll` as above or it MAY share the same command cycle. Simultaneous `poll` calls for the same CCSpec that specify no parameters SHALL share the same command cycle, as implied by the state diagrams in this section.

5.6.2 Lifecycle State Transitions for EC/CCSpecs Created by the Immediate Method

An EC/CCSpec that is created by a call to the `immediate` method of the ALE Reading/Writing API SHALL begin in the *requested* state if any start triggers are specified, and in the *active* state if no start triggers are specified (in this case, an event/command cycle begins immediately). Thereafter, it is subject to state transitions that occur in response to the following kinds of events:

- The `immediate` call being aborted, as provided for in Sections 8.1 and 9.1.
- Event/command cycle stopping conditions, as specified by the EC/CCSpec. The EC/CCSpec parameters that determine starting and stopping conditions are defined in Sections 8.2.1 and 9.3.1.

The state transitions are illustrated in the diagram below. For clarity, not all details of each state transition are shown in the diagram; the tables following the diagram constitute the normative specification of all state transitions.



An EC/CCSpec that is created by a call to the `immediate` method SHALL be subject to the state transitions specified in the two tables below, which are a simplified subset of the tables in Section 5.6.1. In these tables, “start triggers” refer to start condition information that is derived from the EC/CCSpec as described normatively in Sections 8.2.1 and 9.3.1. It is possible for an EC/CCSpec to specify no start triggers, and this figures into the description of the state transitions.

The following transitions SHALL apply when the EC/CCSpec is in the *requested* state:

Event (when in the *requested* state)	**Action**	**Next state**
Arrival of a start trigger	An event/command cycle begins.	*Active*
The `immediate` call is aborted	The event/command cycle ends, with no reports delivered.	(ECSpec no longer exists)

Table 9. State Transitions from the Requested State

The following transitions SHALL apply when the EC/CCSpec is in the *active* state:

Event (when in the *active* state)	**Action**	**Next state**
A stopping condition has occurred as specified in Section 8.2.1 (for an ECSpec) or Section 9.3.1 (for a CCSpec).	The event/command cycle ends. Reports are returned to the `immediate` caller.	(ECSpec no longer exists)

Event (when in the *active* state)	Action	Next state
The `immediate` call is aborted	The event/command cycle ends, with no reports delivered.	(ECSpec no longer exists)

Table 10. State Transitions from the Active State

Events occuring at times other than those specified in the tables above SHALL NOT cause a state transition.

6 Built-in Fieldnames, Datatypes, and Formats

This section defines specific fieldnames, datatypes, and formats that are pre-defined by the ALE specification. These may be used by ALE clients to construct fieldspecs that are used by the Reading API and the Writing API. In addition to those defined here, ALE implementations MAY provide additional pre-defined fieldnames, datatypes, and formats. The Tag Memory API (Section 7) provides a standardized way for ALE clients to define additional fieldnames beyond those pre-defined by an ALE implementation. ALE implementations MAY also provide extension APIs that allow ALE clients to define new datatypes and formats beyond those that are pre-defined.

6.1 Built-in Fieldnames

This section defines fieldnames that are pre-defined by the ALE specification. An ALE implementation SHALL recognize each fieldname defined in this section and interpret it as defined herein. In addition, an ALE implementation that implements the TMSpec API SHALL recognize fieldnames defined through that API (see Section 7).

In general, the definition of a fieldname has to say how it applies to different tag types, and the default datatype and format to be used when not explicitly specified as part of a fieldspec.

6.1.1 The epc fieldname

An ALE implementation SHALL recognize the string `epc` as a valid fieldname as specified in this section.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret the `epc` fieldname as referring to the EPC/UII content of the EPC memory bank (Bank 01_2) as defined in [Gen2]. Specifically, it refers to the toggle bit (bit 17h), the Reserved/AFI bits (bits 18h-1Fh), and the EPC/UII bits (bits 20h through the end of the EPC bank as indicated by the length bits 10h-14h).

When interacting with a Gen1 Tag, an ALE implementation SHALL interpret the `epc` fieldname as referring to the EPC content of the Tag; that is, the EPC payload (the number of bits being fixed by the tag) not including CRC or other non-EPC bits. The treatment SHALL be equivalent to a Gen2 tag whose toggle bit (bit 17h) and Reserved/AFI bits (bits 18h-1Fh) are zeros.

When interacting with a Gen1 or Gen2 Tag, an ALE implementation SHALL raise an "operation not possible" condition if an attempt is made to carry out a "lock" operation on the `epc` field. (The entire EPC bank may be locked, however, using the `epcBank` fieldname defined in Section 6.1.4, below.)

When interacting with any other type of Tag, the interpretation of the `epc` fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The only datatype that may be used with the `epc` fieldname is the `epc` datatype (Section 6.2.1). If a fieldspec specifies a fieldname of `epc` and specifies any other datatype besides `epc`, the ALE implementation SHALL consider the fieldspec to be invalid.

The default datatype for the `epc` fieldname is `epc` (Section 6.2.1). The default format for the `epc` fieldname is `epc-tag` (Section 6.2.1.1).

6.1.2 The `killPwd` fieldname

An ALE implementation SHALL recognize the string `killPwd` as a valid fieldname as specified in this section.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret the `killPwd` fieldname as a synonym for the fieldname `@0.32`, that is, for offset 00_h to $1F_h$ in the RESERVED memory bank of a Gen2 Tag, which holds the Kill Password.

When interacting with any other type of Tag, the interpretation of the `killPwd` fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for the `killPwd` field SHALL be `uint` (Section 6.2.2); the default format SHALL be `hex`. The implementation SHALL NOT permit any other datatypes defined in this specification to be used for the `killPwd` field.

6.1.3 The `accessPwd` fieldname

An ALE implementation SHALL recognize the string `accessPwd` as a valid fieldname as specified in this section.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret the `accessPwd` fieldname as a synonym for the fieldname `@0.32.32`, that is, for offset 20_h to $3F_h$ in the RESERVED memory bank of a Gen2 Tag, which holds the Access Password.

When interacting with any other type of Tag, the interpretation of the `accessPwd` fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for the `accessPwd` field SHALL be `uint` (Section 6.2.2); the default format SHALL be `hex`. The implementation SHALL NOT permit any other datatypes defined in this specification to be used for the `accessPwd` field.

6.1.4 The epcBank fieldname

An ALE implementation SHALL recognize the string `epcBank` as a valid fieldname as specified in this section.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret the `epcBank` fieldname as referring to the content of the EPC memory bank (Bank 01_2) as defined in [Gen2]. Specifically, it refers to the offset 00_h up to the end of this memory bank. When this fieldname is referred by an ALE write command the data is written from offset 00_h till the length of the provided data length. When this fieldname is referred by ALE read command the data is read from offset 00_h through the end of this memory bank. If the implementation cannot or does not wish to support reading to the end of the memory bank, an ALE implementation SHALL raise an “operation not possible” condition when an attempt is made to read from the `epcBank` field.

When interacting with any other type of Tag, the interpretation of the `epcBank` fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for the `epcBank` field SHALL be `bits` (Section 6.2.3); the default format SHALL be `hex`. The implementation SHALL NOT permit any other datatypes defined in this specification to be used for the `epcBank` field.

6.1.5 The `tidBank` fieldname

An ALE implementation SHALL recognize the string `tidBank` as a valid fieldname as specified in this section.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret the `tidBank` fieldname as referring to the content of the TID memory bank (Bank 10_2) as defined in [Gen2]. Specifically, it refers to the offset 00_h up to the end of this memory bank. When this fieldname is referred by an ALE write command the data is written from offset 00_h till the length of the provided data length. When this fieldname is referred by ALE read command the data is read from offset 00_h through the end of this memory bank. If the implementation cannot or does not wish to support reading to the end of the memory bank, an ALE implementation SHALL raise an “operation not possible” condition when an attempt is made to read from the `tidBank` field.

When interacting with any other type of Tag, the interpretation of the `tidBank` fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for the `tidBank` field SHALL be `bits` (Section 6.2.3); the default format SHALL be `hex`. The implementation SHALL NOT permit any other datatypes defined in this specification to be used for the `tidBank` field.

6.1.6 The `userBank` fieldname

An ALE implementation SHALL recognize the string `userBank` as a valid fieldname as specified in this section.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret the `userBank` fieldname as referring to the content of the User memory bank (Bank 11_2) as defined in [Gen2]. Specifically, it refers to the offset 00_h up to the end of this memory bank. When this fieldname is referred by an ALE write command the data is written from offset 00_h till the length of the provided data length. When this fieldname is referred by ALE read command the data is read from offset 00_h through the end of this memory bank. If the implementation cannot or does not wish to support reading to the end of the memory bank, an ALE implementation SHALL raise an “operation not possible” condition when an attempt is made to read from the `userBank` field.

When interacting with any other type of Tag, the interpretation of the `userBank` fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for the `userBank` field SHALL be `bits` (Section 6.2.3); the default format SHALL be `hex`. The implementation SHALL NOT permit any other datatypes defined in this specification to be used for the `userBank` field.

6.1.7 The `afi` fieldname

An ALE implementation SHALL recognize the string `afi` as a valid fieldname as specified in this section.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret the `afi` fieldname as a synonym for the fieldname `@1.8.24`, that is, for offset 18_h to $1F_h$ in the EPC/UII memory bank of a Gen2 Tag, which may hold the ISO 15962 Application Family Identifier (AFI). When interacting with a Gen1 Tag, an ALE implementation SHALL interpret the `afi` fieldname as a “field not found”.

When interacting with a Gen2 Tag, an ALE implementation SHALL raise an “operation not possible” condition if an attempt is made to carry out a “lock” operation on the `afi` field. (The entire EPC bank may be locked, however, using the `epcBank` fieldname defined in Section 6.1.4, above.)

When interacting with any other type of Tag, the interpretation of the `afi` fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for the `afi` field SHALL be `uint` (Section 6.2.2); the default format SHALL be `hex`. The implementation SHALL NOT permit any other datatypes defined in this specification to be used for the `afi` field.

6.1.8 The `nsi` fieldname

An ALE implementation SHALL recognize the string `nsi` as a valid fieldname as specified in this section.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret the `nsi` fieldname as a synonym for the fieldname `@1.9.23`, that is, for offset 17_h to $1F_h$ in the EPC/UII memory bank of a Gen2 Tag, which holds the Numbering System Identifier (NSI). When interacting with a Gen1 Tag, an ALE implementation SHALL interpret the `nsi` fieldname as a "field not found".

When interacting with a Gen2 Tag, an ALE implementation SHALL raise an "operation not possible" condition if an attempt is made to carry out a "lock" operation on the `nsi` field. (The entire EPC bank may be locked, however, using the `epcBank` fieldname defined in Section 6.1.4, above.)

When interacting with any other type of Tag, the interpretation of the `nsi` fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for the `nsi` field SHALL be `uint` (Section 6.2.2); the default format SHALL be `hex`. The implementation SHALL NOT permit any other datatypes defined in this specification to be used for the `nsi` field.

6.1.9 Generic Fieldnames

An ALE implementation SHALL recognize any string beginning with an @ character as a valid fieldname as specified by the syntax in the following sub-sections, provided that the string also meets the constraints defined below. An ALE implementation SHALL consider any string beginning with an `@` character but not conforming to any syntax specified herein, or not meeting the constraints defined below, as an invalid fieldname.

6.1.9.1 Absolute Address Fieldnames

An ALE implementation SHALL recognize any string of the form `@`*`bank`*`.`*`length`*`[.`*`offset`*`]` as a valid fieldname as specified in this section, provided that the string also meets the constraints defined below. Fieldnames of this form are referred to herein as "absolute address fieldnames." An ALE implementation SHALL consider any string beginning with an `@` character but not conforming to this syntax, or not meeting the constraints defined below, as an invalid fieldname.

The constraints are as follows. The *`bank`* portion must be `0` or a positive integer with no leading zeros. The *`length`* portion must be a positive integer with no leading zeros. The *`offset`* portion (if specified) must be `0` or a positive integer with no leading zeros.

An ALE implementation SHALL interpret an absolute address fieldname as a fixed field comprising *`length`* contiguous bits starting at offset *`offset`* within memory bank *`bank`*. If *`offset`* is omitted, the ALE implementation SHALL treat the fieldname in the same way as if `offset` were `0`. The precise interpretation of `offset` and `bank` depends on the type of Tag, as follows.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret `bank` as follows:

`bank` value	Meaning (see [Gen2])
0	Reserved bank (Bank 00_2)
1	EPC/UII bank (Bank 01_2)
2	TID bank (Bank 10_2)
3	User bank (Bank 11_2)

Table 11. Bank Values for Absolute Address Fieldnames

Any other `bank` value SHALL result in a "field not found" condition when interacting with a Gen2 Tag. When interacting with a Gen2 Tag, the fieldname SHALL be interpreted as referring to the contiguous field whose most significant bit is `offset` and whose least significant bit is bit (`offset` + `length` – 1), following the addressing convention specified in [Gen2].

When interacting with a Gen1 Tag, an ALE implementation SHALL interpret a `bank` of `0` as referring to the EPC memory of the Tag. Any other `bank` value SHALL result in a "field not found" condition when interacting with a Gen1 Tag. The `offset` field SHALL be interpreted as referring to an offset from the most significant bit of tag memory, and the fieldname SHALL be interpreted as referring to the contiguous field whose most significant bit is `offset` and whose least significant bit is bit (`offset` + `length` – 1), following that addressing convention.

When interacting with any other type of Tag, the interpretation of `bank` and `offset` is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for absolute address fieldnames is `uint` (Section 6.2.2). The default format for absolute address fieldnames is `hex`. The set of legal datatypes for an absolute address fieldname SHALL be the set of datatypes for which binary encoding and decoding is defined, that is, `uint`, `bits`, `epc` and any implementation-specific datatypes that support binary encoding and decoding.

6.1.9.2 Variable Fieldnames

An ALE implementation SHALL recognize any string of the form `@bank.oid` as a valid fieldname as specified in this sub-section, provided that the string also meets the constraints defined below. Fieldnames of this form are referred to herein as "variable fieldnames."

The constraints for this fieldname form are as follows. The `bank` portion must be `0` or a positive integer with no leading zeros. The `oid` portion must be a valid Object Identifier represented in the URN syntax specified in [RFC3061]. An ALE implementation SHALL interpret a variable fieldname as a variable field, referring to an ISO 15962 “data set” whose Object Identifier is *`oid`* and which is encoded in Tag memory using the encoding rules specified in [ISO15962].

The precise interpretation of *`bank`* depends on the type of Tag, as follows.

When interacting with a Gen2 Tag, an ALE implementation SHALL interpret *`bank`* as follows:

`bank` value	Meaning (see [Gen2])
0	Invalid (“field not found” condition)
1	EPC/UII bank (Bank 01_2)
2	Invalid (“field not found” condition)
3	User bank (Bank 11_2)

Table 12. Bank Values for Variable Fieldnames

Any other *`bank`* value SHALL result in a “field not found” condition when interacting with a Gen2 Tag.

When interacting with a Gen1 Tag, an ALE implementation SHALL result in a “field not found” condition when referring to an ISO data set.

An implementation MAY choose not to support variable fieldnames for WRITE operations, in which case an attempt to do so SHALL raise an “operation not possible” condition. An implementation MAY also choose not to support variable fieldnames for READ operations and for the Reading API, in which case an attempt to do so SHALL raise an “operation not possible” condition.

When interacting with any other type of Tag, the interpretation of a variable fieldname is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

The default datatype for ISO data set fieldnames is `iso-15962-string`. The default format for ISO data set fieldnames is `string`.

Explanation (non-normative): ISO 15962 specifies a scheme for encoding an arbitrary collection of variable-length fields into the memory bank of a Tag. Within that specification, the term “data set” is used in the same way the term “field” is used in this specification. The collection of data sets (fields) is encoded by encoding each data set (field) and concatenating the results together. The complete memory image consists of a Data Storage Format Identifier (DSFID) follows by the concatenated encoded data sets. The DSFID includes information that is necessary to decode what follows.

Each encoded data set is conceptually an object consisting of a name/value pair, as follows.

Name: The name of a field is specified by an Object Identifier (OID) [ASN.1].

Value: The value of an ISO 15962 data set is always a character string of characters drawn from the Unicode character set [Unicode]. Applications may enforce particular *syntax constraints on these strings depending on the OID of a field, but these are not known or enforced at the ALE level.*

ISO 15962 defined an efficient compaction and encoding scheme that seeks to minimize the total number of bits consumed while still allowing each data set to be located and operated upon individually. The compaction rules take advantage of such things as several data sets sharing a common OID prefix,a data set value containing only alphabetic characters, and so on. By treating ISO data sets as string-valued fields having names constructed from an OID, the ALE client is insulated from having to know and apply the encoding and compaction rules specified in ISO 15962.

Note that many industry-specific data elements have been assigned standardized OIDs. Examples include:

GS1 Application Identifiers (AIs) correspond to OIDs of the form `urn:oid.1.0.15961.9.AI`*, where* `AI` *is the application identifier.*

ANSI Data Identifiers (DIs) correspond to OIDs of the form `urn:oid.1.0.15961.10.DI`*, where* `DI` *is the data identifier.*

The International Air Transport Association (IATA) has defined a standard repertoire of data sets having OIDs that begin with the prefix `urn:oid:1.0.15961.12`*.*

6.1.9.3 Variable Pattern Fieldnames

An ALE implementation SHALL recognize variable pattern fieldnames as specified in this section. A variable pattern fieldname has the form `@bank.oid-prefix.*`, where `bank` is as specified in Section 6.1.9.2, and `oid-prefix` is a string conforming to the URN syntax for OIDs specified in [RFC3061].

When an `ECReportOutputFieldSpec` (Section 8.2.11) includes a variable pattern fieldname, the ALE implementation SHALL report all ISO 15962 data sets in the specified memory bank whose OID has `oid-prefix` as a prefix. The fieldname appearing in the `ECReportMemberField` (Section 8.3.7) instance corresponding to each data set SHALL be a variable fieldname (Section 6.1.9.2) containing the full OID of the data set (unless overridden by a non-null `name` parameter in the `ECReportOutputFieldSpec`).

6.2 Built-in Datatypes and Formats

This section defines datatypes and formats that are pre-defined by the ALE specification. An ALE implementation SHALL recognize each datatype and format defined in this section and interpret it as defined herein.

In general, the specification of each datatype has to say what formats may be used with that datatype. Each format has to say whether it is permissible in both reading and writing contexts or only in reading contexts. A format must define a syntax for literal values, for filter patterns, and for grouping patterns.

An ALE implementation SHALL consider a fieldspec invalid if the format is not compatible with the datatype, or if the format is a read-only format and the fieldspec is being used in a context that requires a read-write format.

6.2.1 The epc datatype

An ALE implementation SHALL recognize the string `epc` as a valid datatype as specified in this section.

The `epc` datatype refers to the space of values defined in the EPCglobal Tag Data Standard [TDS1.3.1]. (An implementation MAY support a later version of the EPCglobal Tag Data Standard, in which case it SHALL provide documentation specifying which version it supports.) Because this includes "raw" EPC values, any bit string of any length may be considered a member of the `epc` datatype. The value space also includes EPC values of the form `urn:epc:raw:`*`N.A.V`*, which can only be encoded in contexts where a toggle bit and AFI are available. The encoding and decoding of the `epc` datatype SHALL be according to the EPCglobal Tag Data Standard [TDS1.3.1] (or later, if applicable).

6.2.1.1 Binary Encoding and Decoding of the EPC Datatype

When reading and writing values of the `epc` datatype in a field that includes a toggle bit and AFI (including the `epc` field as specified in Section 6.1.1), decoding and encoding SHALL take place as specified in Section 6.2.1.2 below.

When reading and writing values of the `epc` datatype in a field that does not include a toggle bit and AFI (including an absolute address field as specified in Section 6.1.9.1), the following rules apply. Decoding SHALL take place as specified in Section 6.2.1.2, using the rules for the case where the toggle bit and the AFI are not available. Encoding SHALL take place using those same rules, with the following modifications:

- If the encoded value has more bits than are available in the specified field, an "out of range" condition SHALL be raised.
- If the encoded value has fewer bits than are available in the specified field, the encoded value SHALL be padded with trailing zero bits to fit. That is, the most significant bit of the encoded value is aligned to the most significant bit of the field, and the least significant bits of the field beyond the encoded value are filled with zeros.
- If the EPC value is of the form `urn:epc:raw:`*`N.A.V`* , an "out of range" condition SHALL be raised (because there is no available toggle and AFI, required for values of this form).

6.2.1.2 EPC datatype Formats

An ALE implementation SHALL recognize the format names specified below and permit their use with the `epc` datatype. The notation "RW" below indicates that the ALE implementation SHALL permit the format in both reading and writing contexts, while the notation "RO" indicates that the ALE implementation SHALL permit the format only in reading contexts.

Format	RO/ RW	Interpretation
`epc-pure`	RO	Values are formatted according to the procedure in Section 5.1 of [TDS1.3.1], or Section 5.3 of [TDS1.3.1] if a toggle bit and AFI are available (as when reading from a Gen2 Tag). If the procedure in Section 5.1 of [TDS1.3.1] results in an error, then the value is formatted as a raw hexadecimal value following Step 20 of the procedure in Section 5.2 of [TDS1.3.1], or following Steps 6 through 8 of the procedure in Section 5.4 of [TDS1.3.1] if a toggle bit and AFI are available and the toggle bit is a one.
`epc-tag`	RW	Values are formatted according to the procedure in Section 5.2 of [TDS1.3.1], or Section 5.4 of [TDS1.3.1] if a toggle bit and AFI are available (as when reading from a Gen2 Tag). For writing, the value to write to the Tag is obtained by following the procedure in Section 5.5 of [TDS1.3.1], or the procedure in Section 5.6 of [TDS1.3.1] when writing to a context where a toggle bit and AFI are usable (as when writing to a Gen2 Tag).
`epc-hex`	RW	Values are formatted according to Step 20 of the procedure in Section 5.2 of [TDS1.3.1], or following Steps 6 through 8 of the procedure in Section 5.4 of [TDS1.3.1] if a toggle bit and AFI are available and the toggle bit is a one. For writing, the value to write the Tag is obtained by following the procedure in Section 5.5 of [TDS1.3.1], or the procedure in Section 5.6 of [TDS1.3.1] when writing to a context where a toggle bit and AFI are usable (as when writing to a Gen2 Tag).
`epc-decimal`	RW	Like `epc-hex`, but the V portion of the URI does not include a leading '`x`'

Table 13. EPC Datatype Formats

6.2.1.3 EPC datatype Pattern Syntax

An ALE implementation SHALL recognize pattern syntax as specified below for each of the formats defined for use with the `epc` datatype.

Format	Pattern Syntax
`epc-pure`	A pattern is a URI conforming to the syntax defined in Section 4.2.4 of [TDS1.3.1]. The ALE implementation SHALL interpret a pattern in this form as matching values of the `epc` datatype following the definition in Section 6 of [TDS1.3.1].

Format	Pattern Syntax
epc-tag	A pattern is a URI conforming to the syntax defined in Section 4.2.3 of [TDS1.3.1]. The ALE implementation SHALL interpret a pattern in this form as matching values of the epc datatype following the definition in Section 6 of [TDS1.3.1].
epc-hex	This format has no pattern syntax.
epc-decimal	This format has no pattern syntax.

Table 14. EPC Datatype Pattern Formats

6.2.1.4 EPC datatype Grouping Pattern Syntax

An ALE implementation SHALL recognize grouping pattern syntax as specified below for each of the formats defined for use with the epc datatype.

Format	Pattern Syntax
epc-pure	A grouping pattern is a URI conforming to the syntax defined in Section 4.2.4 of [TDS1.3.1], extended by allowing the character X in each position where a * character is allowed. The interpretation is defined below.
epc-tag	A pattern is a URI conforming to the syntax defined in Section 4.2.3 of [TDS1.3.1], extended by allowing the character X in each position where a * character is allowed. The interpretation is defined below.
epc-hex	This format has no grouping pattern syntax.
epc-decimal	This format has no grouping pattern syntax.

Table 15. EPC Datatype Grouping Formats

As indicated above, a grouping pattern for an epc format has the same syntax as the corresponding pattern 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 [TDS1.3.1] apply equally to the use of the X character. For example, the following are legal grouping patterns for the epc-tag format:

```
urn:epc:pat:sgtin-96:3.*.*.*
urn:epc:pat:sgtin-96:3.*.X.*
urn:epc:pat:sgtin-96:3.X.*.*
urn:epc:pat:sgtin-96:3.X.X.*
```

But the following are not:

```
urn:epc:pat:sgtin-96:3.*.12345.*
urn:epc:pat:sgtin-96:3.X.12345.*
```

EPC grouping patterns SHALL be interpreted as follows:

Pattern URI Field	Meaning

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.

Table 16. Meaning of EPC Grouping Pattern Field Values

Here are examples of grouping patterns for the `epc-tag` format:

Pattern URI	Meaning
`urn:epc:pat:sgtin-96:X.*.*.*`	groups by filter value (*e.g.*, case/pallet)
`urn:epc:pat:sgtin-96:*.X.*.*`	groups by company prefix
`urn:epc:pat:sgtin-96:*.X.X.*`	groups by company prefix and item reference (i.e., groups by specific product)
`urn:epc:pat:sgtin-96:X.X.X.*`	groups by company prefix, item reference, and filter
`urn:epc:pat:sgtin-96:3.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

Table 17. Examples of EPC Grouping Patterns

The name of a group generated from a grouping pattern is the same as the grouping pattern 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-96:3.0036000.123456.400
urn:epc:tag:sgtin-96:3.0036000.123456.500
urn:epc:tag:sgtin-96:3.0029000.111111.100
urn:epc:tag:sscc-96:3.0012345.31415926
```

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

```
urn:epc:pat:sgtin-96:*.X.*.*
```

would generate the following groups in the report:

Group Name	EPCs in Group

Group Name	EPCs in Group
`urn:epc:pat:sgtin-96:*.0036000.*.*`	`urn:epc:tag:sgtin-96:3.0036000.123456.400` `urn:epc:tag:sgtin-96:3.0036000.123456.500`
`urn:epc:pat:sgtin-96:*.0029000.*.*`	`urn:epc:tag:sgtin-96:3.0029000.111111.100`
[default group]	`urn:epc:tag:sscc-96:3.0012345.31415926`

Table 18. Example EPC Grouping Result

The validation rules for grouping patterns include a test for disjointness (see Section 8.2.9). Disjointness 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 included in the range	Disjoint if the ranges do not overlap

Table 19. EPC Grouping Pattern Disjointedness Test

The relationship of the grouping patterns defined above to the group operator introduced in Section 5.2.1 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* = *field_epc_k*, if *field_i_k* = X; or *field_g_k* = *field_i_k*, otherwise.
- If epc does not match Pat_i for any i, then G(epc) = the default group.

6.2.2 Unsigned Integer (uint) Datatype

An ALE implementation SHALL recognize the string uint as a valid datatype as specified in this section.

The space of values for the datatype uint is the set of non-negative integers.

6.2.2.1 Binary Encoding and Decoding of the Unsigned Integer Datatype

When converting between a sequence of N bits and a value of type uint, the leftmost bit SHALL be considered to be the most significant bit.

If an uint value to be encoded to a sequence of N bits is greater than or equal to 2^N, an "out of range" condition SHALL be raised.

6.2.2.2 Unsigned Integer Datatype Formats

An ALE implementation SHALL recognize hex and decimal as valid formats for the uint datatype, as specified below.

In the hex format, an unsigned integer is represented using the following syntax:

```
HexUnsignedInteger ::= 'x' HexDigit+
HexDigit ::= DecimalDigit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
'a' | 'b' | 'c' | 'd' | 'e' | 'f'
DecimalDigit ::= '0' | NonZeroDigit
```

```
NonZeroDigit ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
'9'
```

For output, the ALE implementation SHALL construct a `HexUnsignedInteger` string with no leading zeros, except that the value zero itself is represented by a single '`0`' digit. The string SHALL NOT contain lowercase letters.

For input, the ALE implementation SHALL accept any `HexUnsignedInteger` string.

In the `decimal` format, an unsigned integer is represented using the following syntax:

```
DecimalUnsignedInteger ::= DecimalDigit+
```

For output, the ALE implementation SHALL construct a `DecimalUnsignedInteger` string with no leading zeros, except that the value zero itself is represented by a single '`0`' digit.

For input, the ALE implementation SHALL accept any `DecimalUnsignedInteger` string.

6.2.2.3 Unsigned Integer Pattern Syntax

An ALE implementation SHALL recognize pattern syntax as specified below for each of the formats defined for use with the `uint` datatype.

In the `hex` format, an unsigned integer pattern is represented using the following syntax:

```
HexUnsignedIntegerPattern ::= HexUnsignedInteger | '*' | '['
HexUnsignedInteger '-' HexUnsignedInteger ']' | '&'
HexUnsignedInteger '=' HexUnsignedInteger
```

In the `decimal` format, an unsigned integer pattern is represented using the following syntax:

```
DecimalUnsignedIntegerPattern ::= DecimalUnsignedInteger | '*' |
'[' DecimalUnsignedInteger '-' DecimalUnsignedInteger ']'
```

An ALE implementation SHALL interpret these patterns as follows for both formats. If a pattern is a single integer value (i.e., `HexUnsignedInteger` or `DecimalUnsignedInteger` as appropriate), the pattern matches a value equal to the pattern. If a pattern is the '`*`' character, the pattern matches any value. If a pattern is in the form `[`*`lo-hi`*`]`, the pattern matches any value between *`lo`* and *`hi`*, inclusive. If a pattern is in the form `&`*`mask`*`=`*`compare`* the pattern matches any value that is equal to *`compare`* after being bitwise and-ed with *`mask`*.

For mask-compare patterns, one additional syntactic constraint applies: each bit in *`compare`* must be 0 if its corresponding bit in *`mask`* is also 0.

6.2.2.4 Unsigned Integer Grouping Pattern Syntax

An ALE implementation SHALL recognize grouping pattern syntax as specified below for each of the formats defined for use with the `uint` datatype.

In the `hex` format, an unsigned integer grouping pattern is represented using the following syntax:

```
HexUnsignedIntegerGroupingPattern ::= HexUnsignedIntegerPattern |
'X'
```

In the `decimal` format, an unsigned integer grouping pattern is represented using the following syntax:

```
DecimalUnsignedIntegerGroupingPattern ::=
DecimalUnsignedIntegerPattern | 'X'
```

Unsigned grouping patterns SHALL be interpreted as follows:

Pattern URI Field	**Meaning**
`X`	Create a different group for each distinct value.
`*`	All values belong to the same group.
`Number`	Only values equal to *Number* will belong to this group.
`[`*`Lo-Hi`*`]`	Only values that fall within the specified range (inclusive) will belong to this group.

Table 20. Unsigned Integer Grouping Pattern Field Values

The name of a group generated from a grouping pattern is the same as the grouping pattern, except that if the grouping pattern was '`X`' then the group name is the actual value.

The validation rules for grouping patterns include a test for disjointness (see Section 8.2.9). Disjointness of two patterns is defined as follows. Let Pat_i and Pat_j be two unsigned integer grouping patterns. Then Pat_i and Pat_j are disjoint according to 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 included in the range	Disjoint if the ranges do not overlap

Table 21. Unsigned Integer Grouping Pattern Disjointedness Test

The relationship of the grouping patterns defined above to the group operator introduced in Section 5.2.1 is defined as follows. Formally, a group operator G is specified by a list of grouping patterns:

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

Then the definition of G(*value*) is as follows.

The integer *value* matches Pat_i if one of the following is true:

- Pat_i = `X`
- Pat_i = `*`
- Pat_i is a number, equal to *value*
- Pat_i is a range `[Lo-Hi]`, and `Lo` ≤ *value* ≤ `Hi`

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

G(value) is then defined as follows:

- If value matches Pat_i for some i, then

  G(*value*) = *value*, if Pat_i = X; G(*value*) = Pat_i, otherwise
- If value does not match Pat_i for any i, then G(*value*) = the default group.

6.2.3 The `bits` Datatype

An ALE implementation SHALL recognize the string `bits` as a valid datatype as specified in this section.

The space of values for the datatype `bits` is the set of all non-empty and finite-length sequences of bits. Note that the length of a `bits` value is significant; values of different lengths are always considered to be different, even if they only differ by the amount of leading or trailing zeros.

6.2.3.1 Binary Encoding and Decoding of the Bits Datatype

When reading a value of type `bits`, the ALE implementation SHALL return the unmodified sequence of bits read from the field.

When writing a value of type bits, the following table SHALL be used based on the number of bits in the of the `bits` value (M) and the number of bits in the field (N):

M > N	The bits value to be written is longer than the available number of bits, so an "out of range" condition SHALL be raised.
M = N	The lengths match exactly; the value SHALL be written without modification.

M < N	The field is longer than the value. The value SHALL be written to the leftmost M bits of the destination. (That is, the most significant bit of the value shall be aligned with the most significant bit position of the field.) The remaining N−M bits SHALL each either be set to 0 or retain their previous value, at the discretion of the implementation.

Table 22. Rules for Writing `bits` Values to Fields of Differing Lengths

Explanation (non-normative): The case M < N only requires writing the entire bits value to the field beginning at the field's leftmost position. The implementation may decide if the remaining part of the field is padded with zero bits or left unchanged. The possibility to leave the remaining part unchanged is provided to enable implementation specific optimization. In particular, with fields of unknown length (e.g. `userBank`) just writing the left bits may be more efficient than first determining the actual length of the field and then writing the remaining part padded with zeros.

6.2.3.2 Bits Datatype Formats

An ALE implementation SHALL recognize `hex` as a valid format for the `bits` datatype.

In the `hex` format, a `bits` value is represented by its length in bits and its bit pattern encoded in hexadecimal, using the following syntax:

```
HexBits ::= DecimalPositiveInteger ':' HexUnsignedInteger
DecimalPositiveInteger ::= NonZeroDigit DecimalDigit*
HexUnsignedInteger ::= 'x' HexDigit+
HexDigit ::= DecimalDigit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
'a' | 'b' | 'c' | 'd' | 'e' | 'f'
DecimalDigit ::= '0' | NonZeroDigit
NonZeroDigit ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
'9'
```

For output, the ALE implementation SHALL construct the length part without leading zeros. The bit pattern SHALL be represented using N `HexDigit` characters, where N is the length divided by 4 and rounded up to the next higher integer, padding with leading zero bits as necessary. The string SHALL NOT contain lowercase letters.

For input, the ALE implementation SHALL accept any `HexBits` string where the length specified in the first part of the `HexBits` string, divided by 4 and rounded up to the next higher integer, matches the number of `HexDigit` characters in the second part. If the length is not divisible by 4, the ALE implementation SHALL require the input to be padded with leading zero bits.

6.2.3.3 Bits Pattern Syntax

No pattern syntax is defined for the `hex` format of the `bits` datatype.

6.2.3.4 Bits Grouping Pattern Syntax

No grouping pattern syntax is defined for the `hex` format of the `bits` datatype.

6.2.4 ISO 15962 String Datatype

An ALE implementation SHALL recognize the string `iso-15962-string` as a valid datatype referring to a string of zero or more characters drawn from the Unicode character set [Unicode], encoded according to ISO 15962 [ISO15962].

6.2.4.1 ISO 15962 String Format

An ALE implementation SHALL recognize `string` as a valid format for the `iso-15962-string` datatype. In the `string` format, a string is represented simply as a sequence of Unicode characters corresponding directly to the characters encoded in the Tag.

6.2.4.2 ISO 15962 String Pattern Syntax

No pattern syntax is defined for the `string` format of the `iso-15962-string` datatype.

6.2.4.3 ISO 15962 String Grouping Pattern Syntax

No grouping pattern syntax is defined for the `string` format of the `iso-15962-string` datatype.

7 Tag Memory Specification API

ALE 1.1 provides facilities for filtering, grouping, and reporting of non-EPC tag data. In support of these functions, ALE 1.1 defines fieldspecs (Section 5.4) that specify data fields within Tags, used within the ALE APIs for data contents, filters, and groups.

The structure of user-defined memory fields is likely to be application-specific, so in addition to pre-defined fieldspecs defined in Section 6 the ALE specification provides for user-defined fieldspecs. The API specified in this section provides for user-defined fieldnames that refer to fixed-length, fixed-offset fields that are the same for all Tag types. These user-defined fieldnames are equivalent in functionality to the absolute fixed address fieldnames defined in Section 6.1.9.1 or to the variable fieldnames defined in Section 6.1.9.2. ALE implementations MAY extend this API to provide for definitions of more complex fieldspecs.

An implementation of this API SHALL provide the `TMFixedFieldListSpec` specified in Section 7.3, and SHALL also provide the `TMVariableFieldListSpec` as specified in Section 7.5. An ALE implementation MAY provide other TMSpec as vendor extensions.

7.1 ALETM – Main API class

```
<<interface>>
ALETM
---
defineTMSpec(specName : String, spec : TMSpec) : void
undefineTMSpec(specName : String) : void
getTMSpec(specName : String) : TMSpec
getTMSpecNames() : List<String>
getStandardVersion() : String
getVendorVersion() : String
<<extension point>>
```

An ALE implementation SHALL implement the methods of the ALE Tag Memory Specification API as specified in the following table:

Method	Argument/ Result	Type	Description
`defineTMSpec`	`specName`	`String`	Defines new fieldnames according to `spec`. Thereafter, clients of the Reading and Writing APIs may refer to the fieldnames defined by `spec`. The value of the `specName` parameter is an arbitrary string that the client may use to refer to the TMSpec in subsequent calls to `undefineTMSpec` and `getTMSpec`, but otherwise is not related to the fieldnames defined by the specified `spec`.
	`spec`	`TMSpec`	
	`[result]`	`Void`	
`undefineTMSpec`	`specName`	`String`	Removes the fieldnames defined previously by the TMSpec named `specName`.
	`[result]`	`Void`	
`getTMSpec`	`specName`	`String`	Returns the TMSpec

Method	Argument/ Result	Type	Description
	[result]	TMSpec	previously defined using name specName. The result SHALL be equivalent to the TMSpec that was provided to the define method, but NEED NOT be identical. “Equivalent” means that the returned TMSpec has exactly the same meaning as the original TMSpec when interpreted according to this specification.
getTMSpecNames	[result]	List<String>	Returns an unordered list of the names of previously defined TMSpecs.
getStandardVersion	[result]	String	Returns a string that identifies what version of the specification this implementation of the ALE Tag Memory API complies with, as specified in Section 4.3.
getVendorVersion	[result]	String	Returns a string that identifies what vendor extensions of the ALE Tag Memory API this implementation provides as specified in Section 4.3.

Table 23. ALETM Interface Methods

A tag memory spec, or TMSpec, defines a set of symbolic fieldnames that may be used in fieldspecs within the Reading API and the Writing API. The name of a TMSpec is used as a handle for management of TMSpecs.

7.1.1 Error Conditions

Methods of the Tag Memory Specification 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. If the Tag Memory API implementation is associated with an implementation of the Access Control API (Section 11), the implementation SHALL raise this exception if the client was not granted access rights to the called method as specified in Section 11. Other, implementation-specific circumstances may cause this exception; these are outside the scope of this specification.
`DuplicateNameException`	The specified TMSpec name already exists.
`TMSpecValidationException`	The specified TMSpec is invalid or attempts to define fieldnames that are already defined. The complete list of rules for generating this exception are specified in Sections 7.4 and 7.6.
`NoSuchNameException`	The specified TMSpec name does not exist.
`InUseException`	The specified TMSpec cannot be undefined, because there exist one or more ECSpecs or CCSpecs that refer to it.
`ImplementationException`	A generic exception raised 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.

Table 24. Exceptions for the ALETM Interface

The exceptions that may be raised by each Tag Memory API method are indicated in the table below. An ALE implementation SHALL raise the appropriate exception listed below when the corresponding condition described above occurs. If more than one exception condition applies to a given method call, the ALE implementation may raise any of the exceptions that applies.

ALE Method	Exceptions

`defineTMSpec`	`DuplicateNameException`
	`TMSpecValidationException`
	`SecurityException`
	`ImplementationException`

`undefineTMSpec`	`NoSuchNameException`
	`InUseException`
	`SecurityException`
	`ImplementationException`

`getTMSpec`	`NoSuchNameException`
	`SecurityException`
	`ImplementationException`

`getTMSpecNames`	`SecurityException` `ImplementationException`
`getStandardVersion`	`ImplementationException`
`getVendorVersion`	`ImplementationException`

Table 25. Exceptions Raised by each ALETM Interface Method

7.2 TMSpec (abstract)

`TMSpec` is an abstract class representing any object that an ALE implementation supports as a means to define fieldnames. An ALE implementation SHALL support `TMFixedFieldListSpec` as a possible type of `TMSpec`. An ALE implementation also SHALL support `TMVariableFieldListSpec` as a possible type of `TMSpec`. An ALE implementation MAY provide additional types of `TMSpec`s as vendor extensions to support defining fieldnames in other ways.

For all subtypes of `TMSpec`, the `defineTMSpec` method SHALL raise a `TMSpecValidationException` if any of the following are true:

- Any component of the specified `TMSpec` attempts to create a fieldname that has previously been defined through the Tag Memory Specification API, or is one of the built-in fieldnames specified in Section 6.1. The latter includes any fieldname that begins with the '`@`' character.
- The specified `TMSpec` attempts to create two or more fields with the same fieldname.

Specific subtypes of `TMSpec` MAY specify additional situations under which a `TMSpecValidatonException` is raised.

7.3 TMFixedFieldListSpec

```
TMFixedFieldListSpec
fixedFields : List<TMFixedFieldSpec>
<<extension point>>
---
```

A TMFixedFieldListSpec is a type of TMSpec that defines an unordered list of fieldnames, each fieldname mapping to a specific fixed field described by a bank, offset, and length.

7.4 TMFixedFieldSpec

```
TMFixedFieldSpec
fieldname : String
bank : Integer
length : Integer
offset : Integer
defaultDatatype : String
defaultFormat : String
<<extension point>>
---
```

A TMFixedFieldSpec specifies a single fixed-length field. An ALE implementation SHALL interpret the fields as follows:

Field	Type	Description
fieldname	String	Specifies the symbolic fieldname name that an ALE client may use in a fieldspec to refer to the field defined by this TMFixedFieldSpec.
bank	Integer	Specifies the bank of Tag memory to which fieldname refers. The value of bank SHALL be interpreted by the ALE implementation in the same manner as bank is in the absolute address fieldname (Section 6.1.9.1).
length	Integer	Specifies the length of the contiguous portion of Tag memory to which fieldname refers. The value of length SHALL be interpreted by the ALE implementation in the same manner as length is in the absolute address fieldname (Section 6.1.9.1).

Field	Type	Description
offset	Integer	Specifies the offset within Tag memory of the start of the contiguous portion to which fieldname refers. The value of offset SHALL be interpreted by the ALE implementation in the same manner as offset is in the absolute address fieldname (Section 6.1.9.1).
defaultDatatype	String	Specifies the default datatype for this field. The default datatype is used by the ALE Reading or Writing API when interpreting a fieldspec that omits the datatype parameter.
defaultFormat	String	Specifies the default format for this field. The default format is used by the ALE Reading or Writing API when interpreting a fieldspec that omits the format parameter.

Table 26. TMFixedFieldSpec Fields

The defineTMSpec method SHALL raise a TMSpecValidationException if any of the following are true:

- The value of fieldname is a name that has already been defined through the Tag Memory Specification API, or is one of the built-in fieldnames specified in Section 6.1. The latter includes any fieldname that begins with the '@' character.
- The value of fieldname is the same as the fieldname parameter of another member of the same TMFixedFieldListSpec.
- The value of bank is negative.
- The value of length is zero or negative.
- The value of offset is negative.
- The value of defaultDatatype is not a known datatype, or is not a valid datatype for the specified bank, length, and offset (for example, if the datatype requires more bits than have been provided by length).
- The value of defaultFormat is not a known format, or is not a valid format for the specified defaultDatatype.

7.5 TMVariableFieldListSpec

TMVariableFieldListSpec
variableFields : List<TMVariableFieldSpec>
<<extension point>>

A `TMVariableFieldListSpec` is a type of TMSpec that defines an unordered list of fieldnames, each fieldname mapping to a specific ISO 15962 data set named by an object identifier (OID).

7.6 TMVariableFieldSpec

TMVariableFieldSpec
fieldname : String
bank : Integer
oid : String
<<extension point>>

A TMVariableFieldSpec specifies a variable field (see Section 6.1.9.2 for more information regarding variable fieldnames). This type allows ALE clients to associate a symbolic name with an ISO 15962 object identifier. The associated datatype SHALL be `iso-15962-string` and the format SHALL be `string`. An ALE implementation SHALL interpret the fields as follows:

Field	Type	Description
`fieldname`	`String`	Specifies the symbolic fieldname name that an ALE client may use in a fieldspec to refer to the field defined by this `TMVariableFieldSpec`.
`bank`	`Integer`	Specifies the bank of Tag memory to which `fieldname` refers. The value of `bank` SHALL be interpreted by the ALE implementation in the same manner as `bank` is in the variable fieldname (Section 6.1.9.2).
`oid`	`String`	Specifies the object identifier (OID) of the ISO 15962 data set. This string SHALL be interpreted in the same manner as `oid` is in the variable fieldname (Section 6.1.9.2)..

Table 27. `TMVariableFieldSpec` Fields

The `defineTMSpec` method SHALL raise a `TMSpecValidationException` if any of the following are true:

- The value of `fieldname` is a name that has already been defined through the Tag Memory Specification API, or is one of the built-in fieldnames specified in Section 6.1. The latter includes any fieldname that begins with the '`@`' character.
- The value of `fieldname` is the same as the fieldname parameter of another member of the same `TMVariableFieldListSpec`.
- The value of `bank` is negative.
- The value of `oid` is not valid syntax according to [RFC3061].

8 ALE Reading API

This section defines normatively the ALE Reading API. The external interface is defined by the `ALE` interface (Section 8.1). This interface makes use of a number of complex data types that are documented in the sections following Section 8.1. The specification of the Reading API follows the general rules given in Section 4.

Through the `ALE` interface defined in Section 8.1, clients may define and manage event cycle specifications (ECSpecs), read Tags on-demand by activating ECSpecs synchronously, and enter standing requests (subscriptions) for ECSpecs to be activated asynchronously. Results from standing requests are delivered through the `ALECallback` interface, specified in Section 8.4.

Implementations MAY expose the `ALE` interface of the ALE Reading API via a wire protocol, or via a direct API in which clients call directly into code that implements the API. Likewise, implementations MAY implement the `ALECallback` interface via a wire protocol or via a direct API in which clients receive asynchronous results through a direct callback. This Part I of the ALE 1.1 specification does not define any concrete wire protocol or programming language-specific API, but instead only provides an abstract specification of the interfaces using UML. Part II of the specification [ALE1.1Part2] specifies XML-based wire protocol bindings of the interfaces, including an XSD schema for the API data types, a WS-I compliant WSDL definition of a SOAP binding of the `ALE` interface, and several XML-based bindings of the `ALECallback` interface. Implementations MAY provide additional bindings of the API, including bindings to particular programming languages.

8.1 ALE – Main API Class

```
<<interface>>
ALE
---
define(specName : String, spec : ECSpec) : void
undefine(specName : String) : void
getECSpec(specName : String) : ECSpec
getECSpecNames() : List<String>
subscribe(specName : String, notificationURI : String) : void
unsubscribe(specName : String, notificationURI : String) : void
poll(specName : String) : ECReports
immediate(spec : ECSpec) : ECReports
getSubscribers(specName : String) : List<String>
getStandardVersion() : String
getVendorVersion() : String
<<extension point>>
```

An ALE implementation SHALL implement the methods of the ALE Reading API as specified in the following table:

Method	Argument/ Result	Type	Description
`define`	`specName`	`String`	Creates a new ECSpec having the name `specName`, according to `spec`. The lifecycle of the new ECSpec SHALL be subject to the provisions of Section 5.6.1.
	`spec`	`ECSpec`	
	[result]	`Void`	
`undefine`	`specName`	`String`	Removes the ECSpec named `specName` that was previously created by the `define` method. The effect SHALL be as specified in Section 5.6.1.
	[result]	`Void`	
`getECSpec`	`specName`	`String`	Returns the `ECSpec` that

Method	Argument/ Result	Type	Description
	[result]	`ECSpec`	was provided when the ECSpec named `specName` was created by the `define` method. The result SHALL be equivalent to the ECSpec that was provided to the define method, but NEED NOT be identical. "Equivalent" means that the returned ECSpec has exactly the same meaning as the original ECSpec when interpreted both according to this specification and according to the ALE 1.0 specification.
`getECSpecNames`	[result]	`List<String>`	Returns an unordered list of the names of all ECSpecs that are visible to the caller. The order of this list is implementation-dependent.
`subscribe`	`specName`	`String`	Adds a subscriber having the specified `notificationURI` to the set of current subscribers of the ECSpec named `specName`. The effect SHALL be as specified in Section 5.6.1. The `notificationURI` parameter both identifies a specific binding of the `ALECallback` interface and specifies addressing information meaningful to that binding. See Part II.
	`notifi-cationURI`	`String`	
	[result]	`void`	
`unsubscribe`	`specName`	`String`	Removes a subscriber having the specified `notificationURI` from
	`notify-cationURI`	`String`	

Method	Argument/ Result	Type	Description
	[result]	`void`	the set of current subscribers of the ECSpec named `specName`. The effect SHALL be as specified in Section 5.6.1.
`poll`	`specName`	`String`	Requests an activation of the ECSpec named `specName`, returning the results from the next event cycle to complete, as specified in Section 5.6.1. The ALE implementation MAY provide a means to abort an outstanding `poll` call, by explicit client action, by timeout, or by some other means. If such a means is provided, the effect on the ECSpec lifecycle of aborting the `poll` call SHALL be as specified in Section 5.6.1.
	[result]	`ECReports`	
`immediate`	`spec`	`ECSpec`	Creates an unnamed ECSpec according to spec, and immediately requests its activation. The behavior SHALL be, as specified in Section 5.6.2. The ALE implementation MAY provide a means to abort an outstanding `immediate` call, by explicit client action, by timeout, or by some other means. If such a means is provided, the effect on the ECSpec lifecycle of aborting the `immediate` call SHALL be as specified in Section 5.6.2.
	[result]	`ECReports`	
`getSubscribers`	`specName`	`String`	Returns an unordered,

Method	Argument/ Result	Type	Description
	[result]	List<String>	possibly empty list of the notification URIs corresponding to each of the current subscribers for the ECSpec named specName.
getStandardVersion	[result]	String	Returns a string that identifies what version of the specification this implementation of the Reading API complies with, as specified in Section 4.3.
getVendorVersion	[result]	String	Returns a string that identifies what vendor extensions this implementation of the Reading API provides, as specified in Section 4.3.

Table 28. ALE Interface Methods

The primary data types associated with the ALE Reading 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 subscribers 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.

8.1.1 Error Conditions

Methods of the ALE Reading 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. If the Reading API implementation is associated with an implementation of the Access Control API (Section 11), the Reading API implementation SHALL raise this exception if the client was not granted access rights to the called method as specified in Section 11. Other, implementation-specific circumstances may cause this exception; these are outside the scope of this specification.
`DuplicateNameException`	The specified ECSpec name already exists. Note that the existence of a CCSpec having the same name does *not* cause this exception; ECSpecs and CCSpecs are in different namespaces.
`ECSpecValidationException`	The specified ECSpec is invalid. The complete list of rules for generating this exception is specified in Section 8.2.14.
`InvalidURIException`	The URI specified for a subscriber does not conform to URI syntax as specified in [RFC2396], does not name a binding of the `ALECallback` interface recognized by the implementation, or violates syntax or other rules imposed by a particular binding.
`NoSuchNameException`	The specified ECSpec name does not exist.
`NoSuchSubscriberException`	The specified subscriber does not exist.
`DuplicateSubscriptionException`	The specified ECSpec name and subscriber URI is identical to a previous subscription that was created and not yet unsubscribed.

Exception Name	Meaning
`ImplementationException`	A generic exception raised 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.

Table 29. Exceptions in the ALE Interface

The exceptions that may be raised by each ALE method are indicated in the table below. An ALE implementation SHALL raise the appropriate exception listed below when the corresponding condition described above occurs. If more than one exception condition applies to a given method call, the ALE implementation may raise any of the exceptions that applies.

ALE Method	Exceptions

`define`	`DuplicateNameException`
	`ECSpecValidationException`
	`SecurityException`
	`ImplementationException`

`undefine`	`NoSuchNameException`
	`SecurityException`
	`ImplementationException`

`getECSpec`	`NoSuchNameException`
	`SecurityException`
	`ImplementationException`

`getECSpecNames`	`SecurityException` `ImplementationException`

`subscribe`	`NoSuchNameException`
	`InvalidURIException`
	`DuplicateSubscriptionException`
	`SecurityException`
	`ImplementationException`

`unsubscribe`	`NoSuchNameException`
	`NoSuchSubscriberException`
	`InvalidURIException`
	`SecurityException`
	`ImplementationException`

ALE Method	Exceptions

`poll`	`NoSuchNameException`
	`SecurityException`
	`ImplementationException`

`immediate`	`ECSpecValidationException`
	`SecurityException`
	`ImplementationException`

`getSubscribers`	`NoSuchNameException`
	`SecurityException`
	`ImplementationException`

`getStandardVersion`	`ImplementationException`
`getVendorVersion`	`ImplementationException`

Table 30. Exceptions Raised by each ALE Interface Method

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 data 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.

```
ECSpec
logicalReaders : List<String>    // List of logical reader
names
boundarySpec : ECBoundarySpec
reportSpecs : List<ECReportSpec>
includeSpecInReports : Boolean
primaryKeyFields : List<String> // List of fieldnames
strings
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of an `ECSpec` as follows.

Field	Type	Description
`logicalReaders`	`List<String>`	An unordered list that specifies one or more logical readers that are used to acquire tags.

Field	Type	Description
boundarySpec	ECBoundarySpec	Specifies the starting and stopping conditions for event cycles. See Section 8.2.1.
reportSpecs	List<ECReportSpec>	An ordered list that specifies one or more reports to be included in the output from each event cycle. See Section 8.2.5.
includeSpecInReports	Boolean	If true, specifies that each ECReports instance generated from this ECSpec SHALL include a copy of the ECSpec. If false, each ECReports instance SHALL NOT include a copy of the ECSpec.
primaryKeyFields	List<String>	(Optional) An ordered list that specifies a set of fields which together constitute the "primary key" for determining Tag uniqueness, as described below. Each element of the list is a fieldname. If omitted, the ALE implementation SHALL use only the epc field to determine Tag uniqueness, as described below. This gives back-compatibility with ALE 1.0.

Table 31. ECSpec Fields

The define and immediate methods SHALL raise an ECSpecValidationException if any of the following are true for an ECSpec instance:

- The logicalReaders parameter is null, omitted, is an empty list, or contains any logical reader names that are not known to the implementation.
- The boundarySpec parameter is null or omitted, or the specified boundarySpec leads to an ECSpecValidationException as specified in Section 8.2.1.

- The reportSpecs parameter is null, omitted, empty, or any of the members of reportSpecs leads to an ECSpecValidationException as specified in Section 8.2.5.
- Any member of the specified primaryKeyFields is not a known fieldname.
- The implementation does not support the specified primaryKeyFields value with the specified logical readers. An implementation SHALL NOT, however, raise the exception if primaryKeyFields is omitted or its value is a list consisting of the single element epc.

The primaryKeyFields parameter is a list of strings, each one of which is a fieldname naming a field that contributes to a "primary key" for determining Tag uniqueness. As an ALE implementation accumulates Tags during an event cycle, the implementation SHALL consider two Tags to be the same if both tags have the exact same values in all of the primary key fields. The ALE implementation SHALL also use the same rule to determine equality in implementing the ADDITIONS and DELETIONS values of ECReportSetSpec (Section 8.2.6) and the reportOnlyOnChange feature of ECReportSpec (Section 8.2.5). If accessing any of the primary key fields on a Tag causes a "field not found" or "operation not possible" condition, then that Tag SHALL be omitted from the event cycle. If the primaryKeyFields parameter is empty or omitted, the ALE implementation SHALL behave as though primaryKeyFields was set to a list containing the single element epc (this gives behavior compatible with ALE 1.0).

Explanation (non-normative): The primaryKeyFields parameter allows an implementation to optimize its interaction with Tags, because the implementation may avoid reading fields of a Tag if its primary key fields are recognized to be identical to a previously read Tag. The client application must set primaryKeyFields based on its knowledge that (a) only one Tag with a given set of primary key values will be visible within any given event cycle; or (b) multiple Tags having identical primary key values will also have identical values for any other fields relevant to the ECSpec; or (c) if multiple Tags have identical primary key values, the values read from any one such Tag or combination of such Tags are acceptable to the application. If an implementation encounters two or more Tags having identical primary key values within the same event cycle, the implementation is free to use any one or any combination of those Tags to supply the values for other fields that are needed by the ECSpec. For example, an implementation may choose to randomly pick which tag to retrieve the data from, or it may pick the first or last tag seen, and so forth.

Because some Readers may implicitly perform duplicate removal using a fixed set of primary key fields, it may not be possible to implement a given primaryKeyFields *setting for a given logical reader. For this reason, an implementation may raise* ECSpecValidationException *if the* primaryKeyFields *setting cannot be implemented.*

8.2.1 ECBoundarySpec

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

```
ECBoundarySpec
startTrigger : ECTrigger  // deprecated
startTriggerList : List<ECTrigger>
repeatPeriod : ECTime
stopTrigger : ECTrigger // deprecated
stopTriggerList : List<ECTrigger>
duration : ECTime
stableSetInterval : ECTime
whenDataAvailable : Boolean
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of an `ECBoundarySpec` as follows.

Field	Type	Description
`startTrigger`	`ECTrigger`	(Optional) This parameter is deprecated in ALE 1.1, and is provided for back-compatibility with ALE 1.0. If the `startTrigger` parameter is specified with value *T*, the ALE implementation SHALL treat it in the same way as if the `startTriggerList` parameter included *T* as one of its members.
`startTriggerList`	`List<ECTrigger>`	(Optional) An unordered list that specifies zero or more triggers that may start a new event cycle for this `ECSpec`.
`repeatPeriod`	`ECTime`	(Optional) Specifies an interval of time for starting a new event cycle for this `ECSpec`, relative to the start of the previous event cycle.

Field	Type	Description
stopTrigger	ECTrigger	(Optional) This parameter is deprecated in ALE 1.1, and is provided for back-compatibility with ALE 1.0. If the stopTrigger parameter is specified with value *T*, the ALE implementation SHALL treat it in the same way as if the stopTriggerList parameter included *T* as one of its members.
stopTriggerList	List<ECTrigger>	(Optional) An unordered list that specifies zero or more triggers that may stop an event cycle for this ECSpec.
duration	ECTime	(Optional) Specifies an interval of time for stopping an event cycle for this ECSpec, relative to the start of the event cycle. If omitted or equal to zero, has no effect on the stopping of the event cycle.
stableSetInterval	ECTime	(Optional) Specifies that an event cycle may be stopped if no new tags are read within the specified interval. If omitted or equal to zero, has no effect on the stopping of the event cycle.
whenDataAvailable	Boolean	(Optional) If true, specifies that an event cycle may be stopped when any Tag is read that matches the filter conditions of at least one ECReportSpec within this ECSpec. If omitted or false, has no effect on the stopping of the event cycle.

Table 32. ECBoundarySpec Fields

The define and immediate methods SHALL raise an ECSpecValidationException if any of the following are true for an ECBoundarySpec instance:

- A negative number is specified for any of the `ECTime` values `duration`, `repeatPeriod`, and `stableSetInterval`.

- The value of the `startTrigger` or `stopTrigger`, or any element of `startTriggerList` or `stopTriggerList` does not conform to URI syntax as defined by [RFC2396], or is a URI that is not supported by the ALE implementation. Note that an empty string does not conform to URI syntax as defined by [RFC2396].

- No stopping condition is specified; *i.e.*, `stopTrigger` is omitted or null, `stopTriggerList` is empty, `duration` is zero or omitted, `stableSetInterval` is zero or omitted, `whenDataAvailable` is false, and no vendor extension stopping condition is specified.

In the description below, the phrase "the set of start triggers" refers to all start triggers specified in the `startTrigger` and `startTriggerList` parameters, excluding nulls and empty strings. Likewise, the phrase "the set of stop triggers" refers to all stop triggers specified in the `stopTrigger` and `stopTriggerList` parameters, excluding nulls and empty strings. The phrase "if specified" used in reference to `repeatPeriod`, `duration`, or `stableSetInterval` means that the parameter is specified and is a positive (non-zero) number.

The `boundarySpec` parameter of `ECSpec` (of type `ECBoundarySpec)` specifies starting and stopping conditions as referred to in the ECSpec lifecycle specified in Sections 5.6.1 and 5.6.2. Within that description, "arrival of a start trigger" means that the ALE implementation receives any of the triggers specified in the set of start triggers for this ECSpec, and "repeat period" means the value of the `repeatPeriod` parameter, if specified. The phrase "a stopping condition has occurred" means the first of the following to occur:

- The `duration`, when specified, expires (measured from the start of the event cycle).

- When the `stableSetInterval` is specified, no *new* Tags are read by any Reader for the specified interval (*i.e.*, the set of Tags 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 reader cycle containing Tag X, then subsequently Reader B completes a different reader cycle containing the same Tag X, then the occurrence of Tag X in B's reader 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* Tags are detected; it does not imply that previously detected Tags must continue to be detected. That is, only *additions*, and not *deletions*, are considered in determining that the Tag set is "stable."

- Any one of the stop triggers specified in the set of stop triggers is received.

- The `whenDataAvailable` parameter is true, and any Tag is read that matches the filter conditions of at least one `ECReportSpec` within this `ECSpec`. If several matching Tags are read in a single reader cycle, the implementation MAY terminate the event cycle after receiving all of those Tags (that is, the implementation does not have to consider only one of those Tags as terminating the event cycle, saving the others for future event cycles).

Explanation (non-normative) An event cycle begins when the first start condition (repeat period or one of the start triggers) occurs. If no start triggers are specified, the first event cycle begins immediately after the ECSpec becomes requested, otherwise the ECSpec waits in the requested *state until a trigger arrives. Thereafter, if neither a repeat period or any start triggers are specified, another event cycle begins immediately after the prior one ends.*

Also, 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 stopping 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).

Likewise, an event cycle ends when the first stopping condition occurs. 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 Tags is stable for `stableSetInterval`, the event cycle terminates even if the total time is shorter than the specified `duration`.

Start conditions have no effect while an event cycle is active, nor do stopping conditions have an effect when an event cycle is not in progress. For example, if a second start trigger is received while an event cycle is active, it has no effect. For this reason, if a given start trigger is specified twice, it has the same effect as if it were specified only once.

8.2.2 ECTime

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

ECTime
`duration : Long`
`unit : ECTimeUnit`
`---`

The ALE implementation SHALL interpret the fields of an `ECTime` instance as follows.

Field	Type	Description
`duration`	`Long`	The amount of time, in units specified by `unit`.

Field	Type	Description
unit	ECTimeUnit	The unit of time represented by one unit of duration.

Table 33. ECTime Fields

Note that ECTime is used both by the Reading API and the Writing API. Unless otherwise noted, the interpretation of an ECTime instance is the same in both APIs.

8.2.3 ECTimeUnit

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

<<Enumerated Type>> ECTimeUnit
MS
<<extension point>>

The ALE implementation SHALL interpret an instance of ECTimeUnit as specified in the following table.

ECTimeUnit	Unit of Time of duration field of ECTime
MS	Milliseconds

Table 34. ECTimeUnit Fields

Note that ECTimeUnit is used both by the Reading API and the Writing API. Unless otherwise noted, the interpretation of an ECTimeUnit instance is the same in both APIs.

8.2.4 ECTrigger

ECTrigger denotes a URI that is used to specify a start or stop trigger for an event cycle or command cycle (see Section 5.6 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. URIs that begin with the string urn:epcglobal:, however, are reserved for standardized trigger URIs whose meaning is governed by this or other EPCglobal specifications.

Not all URIs beginning with urn:epcglobal: are valid trigger URIs. An implementation SHALL raise an ECSpecValidationException if presented with a URI beginning with urn:epcglobal: that is not valid according to this specification or any other EPCglobal specification that defines a standardized trigger URI. Not all URIs specified in EPCglobal specifications are required to be implemented. An implementation MAY raise an ECSpecValidationException if presented with a URI beginning with urn:epcglobal: that the implementation chooses not to support.

Otherwise, the implementation SHALL interpret the URI according to the relevant specification.

URIs not beginning with `urn:epcglobal:` MAY be interpreted by an implementation in an implementation-dependent manner. If such a URI is not valid according to the implementation-specific rules, the implementation SHALL raise an `ECSpecValidationException`.

Note that `ECTrigger` is used both by the Reading API and the Writing API. Unless otherwise noted, the interpretation of an `ECTrigger` instance is the same in both APIs.

8.2.4.1 Real-time Clock Standardized Trigger

URIs beginning with the string `urn:epcglobal:ale:trigger:rtc:` are reserved for triggers as specified below. An ALE implementation MAY provide support for trigger URIs of this form; if it does, the ALE implementation SHALL conform to the following specification for all such URIs valid according to the specification below.

A real-time clock trigger takes one of the two following forms:

`urn:epcglobal:ale:trigger:rtc:`***`period.offset`***
`urn:epcglobal:ale:trigger:rtc:`***`period.offset.timezone`***

where ***`period`***, ***`offset`***, and ***`timezone`*** are as specified below.

Field	Syntax	Meaning
`period`	A decimal integer numeral in the range 1 ≤ ***`period`*** ≤ 86400000.	The period, in milliseconds, between consecutive triggers occurring within one day. See below.
`offset`	A decimal integer numeral greater than or equal to zero and less than the specified ***`period`***.	The interval, in milliseconds, between midnight and the first trigger delivered after midnight. See below.
`timezone`	A time zone offset specifier having one of the three following forms: `+`***`hh`***`:`***`mm`*** `-`***`hh`***`:`***`mm`*** `Z` Where ***`h`*** and ***`m`*** each denote a single decimal digit.	The time zone in which to interpret "midnight" in the specification of the trigger timing below. `+`***`hh`***`:`***`mm`*** indicates a positive offset (in hours and minutes) from UTC, `-`***`hh`***`:`***`mm`*** indicates a negative offset from UTC, and Z indicates a zero offset from UTC.

Table 35. Real-time Clock Trigger URI Fields

If an ALE implementation chooses to implement triggers of this form, it SHALL interpret a trigger of this form as follows. The trigger is delivered each time the number of milliseconds past midnight modulo ***`period`*** equals ***`offset`***. "Midnight" refers to midnight in the specified time zone, which if omitted defaults to some implementation-dependent default value (typically the time zone configured in the operating system or other platform in which the ALE implementation is running).

Example (non-normative) The following trigger URI denotes a trigger that occurs every hour on the hour:

`urn:epcglobal:ale:trigger:rtc:3600000.0`

The following two trigger URIs denote a pair of one-minute triggers that alternate. Each trigger occurs 30 seconds after the other trigger.

```
urn:epcglobal:ale:trigger:rtc:60000.0
urn:epcglobal:ale:trigger:rtc:60000.30000
```

Note that if the specified `period` does not divide evenly into the number of milliseconds in a day (86,400,000), then the trigger will not be perfectly periodic, because the pattern will be realigned to the specified `offset` each day at midnight.

8.2.5 ECReportSpec

An `ECReportSpec` specifies one report to be included in the list of reports that results from executing an event cycle. An `ECSpec` contains a list of one or more `ECReportSpec` instances. When an event cycle completes, an `ECReports` instance is generated, unless suppressed as described below. An `ECReports` instance contains one or more `ECReport` instances, each corresponding to an `ECReportSpec` instance in the `ECSpec` that governed the event cycle. The number of `ECReport` instances may be fewer than the number of `ECReportSpec` instances, due to the rules for suppression of individual `ECReport` instances as described below.

ECReportSpec
reportName : String
reportSet : ECReportSetSpec
filterSpec : ECFilterSpec
groupSpec : ECGroupSpec
output : ECReportOutputSpec
reportIfEmpty : Boolean
reportOnlyOnChange : Boolean
statProfileNames : List<ECStatProfileName>
<<extension point>>

The ALE implementation SHALL interpret the fields of an `ECReportSpec` as follows.

Field	Type	Description
reportName	String	Specifies a name for reports generated from this ECReportSpec. The ALE implementation SHALL copy this name into the ECReport instance generated from this ECReportSpec.
reportSet	ECReportSetSpec	Specifies what set of Tags are considered for reporting: CURRENT, ADDITIONS, or DELETIONS as described in Section 8.2.6.
filterSpec	ECFilterSpec	Specifies how Tags are filtered before inclusion in the report, as specified in Section 8.2.7.
groupSpec	ECGroupSpec	Specifies how filtered Tags are grouped together for reporting, as specified in Section 8.2.9.
output	ECReportOutputSpec	Specifies which fields to report from each Tag or a count, or both, as specified in Section 8.2.10.
reportIfEmpty	Boolean	Specifies whether to omit the ECReport instance if the final set of Tags is empty, as specified below.
reportOnlyOnChange	Boolean	Specifies whether to omit the ECReport instance if the set of filtered Tags is unchanged from the previous event cycle, as specified below.
statProfileNames	List<ECStatProfileName>	An ordered list that specifies zero or more statistics profiles that govern what statistics are to be included in the report, as specified in Section 8.3.9.

Table 36. ECReportSpec Fields

The define and immediate methods SHALL raise an ECSpecValidationException if any of the following are true for an ECReportSpec instance:

- The specified `reportName` is an empty string or is not accepted by the implementation according to Section 4.5.
- The specified `reportName` is a duplicate of another report name in the same `ECSpec`.
- The specified `filterSpec` leads to an `ECSpecValidationException` as specified in Section 8.2.7.
- The specified `groupSpec` leads to an `ECSpecValidationException` as specified in Section 8.2.9.
- The specified `output` leads to an `ECSpecValidationException` as specified in Section 8.2.10.
- Any element of `statProfileNames` is not the name of a known statistics profile.

An ECReports instance SHALL include an ECReport instance corresponding to each ECReportSpec in the governing ECSpec, in the same order specified in the ECSpec, except that an ECReport instance SHALL be omitted under the following circumstances:

- 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 Tags is empty (i.e., if the final Tag 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 Tags is identical to the filtered prior set of Tags, where equality is tested by considering the `primaryKeyFields` as specified in the `ECSpec` (see Section 8.2), and where the phrase 'the prior set of Tags' is as defined in Section 8.2.6. 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 parameter, or the `reportIfEmpty` parameter.

When the processing of `reportIfEmpty` and `reportOnlyOnChange` results in *all* `ECReport` instances being omitted from an `ECReports` for an event cycle, then the delivery of results to subscribers SHALL be suppressed altogether. That is, a result 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.

Explanation (non-normative): 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.*

The `statProfileNames` parameter is a list of `ECStatProfileName`, each of which corresponds to a statistics profile that will be included in the `ECReports`. If the ALE engine does not recognize any name in the list it SHALL raise an `ECSpecValidationException`.

8.2.6 ECReportSetSpec

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

<<Enumerated Type>> ECReportSetSpec
CURRENT
ADDITIONS
DELETIONS
<<extension point>>

An ALE implementation SHALL interpret an instance of `ECReportSetSpec` as specified in the following table:

ECReportSetSpec value	**Meaning**
CURRENT	The set of tags considered for filtering and output SHALL be the set of Tags read during the event cycle.
ADDITIONS	The set of tags considered for filtering and output SHALL be the set of Tags read during the event cycle, minus the prior set of Tags; that is, the set of Tags that were read during the event cycle and not members of the prior set of Tags. The meaning of "the prior set of Tags" is specified below.
DELETIONS	The set of tags considered for filtering and output SHALL be the prior set of Tags, minus the set of Tags read during the event cycle; that is, the set of Tags that were not read during the event cycle but are members of the prior set of Tags. The meaning of "the prior set of Tags" is specified below.

Table 37. `ECReportSetSpec` Values

The meaning of “the prior set of Tags” is as follows. For a given subscriber to an ECSpec, beginning with the second event cycle to be completed after the `subscribe` call, the prior set of Tags SHALL refer to the set of Tags read during the immediately previous event cycle for that ECSpec. For the first event cycle to be completed after the `subscribe` call for a given subscriber, and for a `poll` call, the prior set of Tags SHALL refer to either the set of Tags read during some previous event cycle for that ECSpec, or the empty set, at the discretion of the implementation. An ALE implementation SHOULD provide documentation that specifies its behavior in these cases.

8.2.7 ECFilterSpec

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

```
ECFilterSpec
includePatterns : List<String>    // List of EPC patterns
(deprecated)
excludePatterns : List<String>    // List of EPC patterns
(deprecated)
filterList : List<ECFilterListMember>
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of an `ECFilterSpec` as follows.

Field	Type	Description
`includePatterns`	`List<String>`	This parameter is deprecated in ALE 1.1, and is provided for back-compatibility with ALE 1.0. If the `includePatterns` parameter is specified with pattern list *L*, the ALE implementation SHALL treat it in the same way as if the `includePatterns` parameter were omitted and `filterList` included an `ECFilterListMember` whose `includeExclude` parameter is set to `INCLUDE`, whose `fieldspec` parameter is set to an `ECFieldSpec` instance whose `fieldname` parameter is set to `epc` and whose `datatype` and `format` parameters are omitted, and whose `patList` parameter is set to *L*.

Field	Type	Description
excludePatterns	List<String>	This parameter is deprecated in ALE 1.1, and is provided for back-compatibility with ALE 1.0. If the `excludePatterns` parameter is specified with pattern list *L*, the ALE implementation SHALL treat it in the same way as if the `excludePatterns` parameter were omitted and `filterList` included an `ECFilterListMember` whose `includeExclude` parameter is set to `EXCLUDE`, whose `fieldspec` parameter is set to an `ECFieldSpec` instance whose `fieldname` parameter is set to `epc` and whose `datatype` and `format` parameters are omitted, and whose `patList` parameter is set to *L*.
filterList	List<ECFilter ListMember>	Specifies an unordered list of filters, as specified below.

Table 38. `ECFilterSpec` Fields

The `define` and `immediate` methods SHALL raise an `ECSpecValidationException` if any of the following are true for an `ECFilterSpec` instance:

- Any element of `includePatterns` is not a syntactically valid `epc-tag` pattern as specified in Section 6.2.1.3.
- Any element of `excludePatterns` is not a syntactically valid `epc-tag` pattern as specified in Section 6.2.1.3.
- Any element of `filterList` leads to an `ECSpecValidationException` as specified in Section 8.2.8.

The `ECFilterSpec` implements a flexible filtering scheme based on a list of `ECFilterListMember` instances. Each `ECFilterListMember` instance defines a test to be applied to fields of a Tag to determine if the Tag should be included in the report. A Tag SHALL be included in the final report if it passes the test specified by *every* `ECFilterListMember` in `filterList`, as defined below.

Each `ECFilterListMember` specifies either an inclusive or an exclusive test based on the value of one field of a Tag. If the `includeExclude` parameter of an `ECFilterListMember` is `INCLUDE`, then the Tag passes the test if and only if accessing the field does not cause a "field not found" or "operation not possible" condition and the value of the field matches at least one pattern specified in the

ECFilterListMember instance. If the includeExclude parameter of an ECFilterListMember is EXCLUDE, then the Tag passes the test if and only if accessing the field causes a “field not found” or “operation not possible” condition or the value of the field does not match any pattern specified in the ECFilterListMember instance.

This can be expressed using the notation of Section 5 as follows, where R is the set of Tags to be reported from a given event cycle, prior to filtering:

$$
\begin{aligned}
F(R) = \{\, tag \mid\ & tag \in R \\
& \&\ (tag \in I_{1,1} \mid \ldots \mid tag \in I_{1,n}) \\
& \&\ (tag \in I_{2,1} \mid \ldots \mid tag \in I_{2,n}) \\
& \&\ \ldots \\
& \&\ (tag \notin E_{1,1} \ \&\ \ldots \ \&\ tag \notin E_{1,n}) \\
& \&\ (tag \notin E_{2,1} \ \&\ \ldots \ \&\ tag \notin E_{2,n}) \\
& \&\ \ldots \,\}
\end{aligned}
$$

where $I_{i,j}$ denotes the set of Tags matched by the *j*th pattern in the patList of the *i*th member of filterList whose includeExclude flag is set to INCLUDE, and $E_{i,j}$ denotes the set of Tags matched by the *j*th pattern in the patList of the *i*th member of filterList whose includeExclude flag is set to EXCLUDE. For the purposes of this definition, includePatterns and excludePatterns are to be treated as though they were additional entries in filterList, as described in the definition of those two parameters in the table above.

8.2.8 ECFilterListMember

An ECFilterListMember specifies filtering by comparing a single field of a Tag to a set of patterns. This type is used in both the Reading API and the Writing API.

```
                     ECFilterListMember
includeExclude : ECIncludeExclude // (INCLUDE or EXCLUDE)
fieldspec : ECFieldSpec
patList : List<String>  // one or more patterns
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of an ECFilterListMember as follows.

Field	Type	Description
includeExclude	ECIncludeExclude	Specifies whether this ECFilterListMember is inclusive or exclusive. If this parameter is INCLUDE, a Tag is considered to pass the filter if the value in the specified field matches any of the patterns in patList. If this parameter is EXCLUDE, a Tag is considered to pass the filter it the value in the specified field does not match any of the patterns in patList.
fieldspec	ECFieldSpec	Specifies which field of the Tag is considered in evaluating this filter, the datatype of the field contents, and the format for patterns that appear in patList.
patList	List<String>	An unordered list that specifies the patterns against which the value of the specified Tag field is to be compared. Each member of this list is a pattern value conforming to the format implied by fieldspec.

Table 39. ECFilterListMember Instances

The define and immediate methods SHALL raise an ECSpecValidationException or CCSpecValidationException (in the Reading API or the Writing API, respectively) if any of the following are true for any ECFilterListMember instance:

- The specified fieldspec is invalid (see Section 8.2.12).
- The patList is empty.
- Any element of patList does not conform to the syntax rules for patterns implied by the specified fieldspec.

8.2.9 ECGroupSpec

ECGroupSpec defines how filtered EPCs are grouped together for reporting.

ECGroupSpec
fieldspec : ECFieldSpec
patternList : List<String> // of pattern URIs
<<extension point>>

The ALE implementation SHALL interpret the fields of an `ECGroupSpec` as follows.

Field	Type	Description
`fieldspec`	`ECFieldSpec`	(Optional) Specifies which field of the Tag is used for grouping, the datatype of the field contents, and the format for grouping patterns that appear in `patternList`. If this parameter is omitted, the ALE implementation SHALL behave as though the `fieldspec` parameter were set to an `ECFieldSpec` instance whose `fieldname` parameter is set to `epc` and whose `datatype` and `format` parameters are omitted.
`patternList`	`List<String>`	An unordered list that specifies the grouping patterns used to generate a group name from the value of the specified Tag field. Each member of this list is a grouping pattern value conforming to the format implied by `fieldspec`.

Table 40. `ECGroupSpec` Fields

The `define` and `immediate` methods SHALL raise an `ECSpecValidationException` if any of the following are true for an `ECGroupSpec` instance:

- The specified `fieldspec` is invalid (see Section 8.2.12).
- The specified `fieldspec` implies a datatype and format for which no grouping pattern syntax is defined.
- Any element of `patternList` does not conform to the syntax rules for grouping patterns implied by the specified `fieldspec`.
- The elements of `patternList` are not disjoint, according to the definition of disjointedness defined by the datatype and format implied by the specified `fieldspec`.

Every filtered Tag that is part of an event cycle SHALL be assigned to exactly one group for purposes of reporting. The group name is determined by the value of the field specified by `fieldspec`, in the following manner. If the field value matches one of the grouping patterns in `patternList`, the group name SHALL be computed from the field value according to the formula specified in the definition of the datatype and format implied by fieldspec. If the field value does not match any of the grouping patterns in `patternList`, or if accessing the field causes a “field not found” or “operatio not possible” condition, the Tag SHALL be assigned to a special “default group.” The name of the default group SHALL be null. Note that a Tag cannot match more than one grouping pattern in `patternList` because of the disjointedness constraint.

If the pattern list is empty (or if the `group` parameter of the `ECReportSpec` is null or omitted), then all Tags SHALL be assigned to the default group.

8.2.10 ECReportOutputSpec

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

ECReportOutputSpec
includeEPC : Boolean
includeTag : Boolean
includeRawHex : Boolean
includeRawDecimal : Boolean
includeCount : Boolean
fieldList : List<ECReportOutputFieldSpec>
<<extension point>>

The parameters of `ECReportOutputSpec` determine which parameters are present in each `ECReportGroup` instance that appears as part of an `ECReport` generated from this `ECReportSpec`. If any of `includeEPC`, `includeTag`, `includeRawHex`, or `includeRawDecimal` are true, or if `fieldList` is non-empty, the ALE implementation SHALL set the `groupList` parameter of each `ECReportGroup` instance to an `ECReportGroupList` instance, which in turn SHALL contain a list of `ECReportGroupListMember` instances having parameters set according to the table below. Otherwise, the ALE implementation SHALL set the `groupList` parameter to null. If `includeCount` is true, the ALE implementation SHALL set the `groupCount` parameter of each `ECReportGroup` instance to an `ECReportGroupCount` instance, with parameters set according to the table below. Otherwise, the ALE implementation SHALL set the `groupCount` parameter to null.

Field	Type	Description
`includeEPC`	`Boolean`	If true, each generated `ECReportGroupListMember` instance SHALL include an `epc` parameter containing the value of the `epc` field of the Tag represented in the `epc-pure` format. If false, each `ECReportGroupListMember` SHALL NOT include the `epc` parameter.
`includeTag`	`Boolean`	If true, each generated `ECReportGroupListMember` instance SHALL include a `tag` parameter containing the value of the `epc` field of the Tag represented in the `epc-tag` format. If false, each `ECReportGroupListMember` SHALL NOT include the `tag` parameter.
`includeRawHex`	`Boolean`	If true, each generated `ECReportGroupListMember` instance SHALL include a `rawHex` parameter containing the value of the `epc` field of the Tag represented in the `epc-hex` format. If false, each `ECReportGroupListMember` SHALL NOT include the `rawHex` parameter.

Field	Type	Description
`includeRawDecimal`	`Boolean`	If true, each generated `ECReportGroupListMember` instance SHALL include a `rawDecimal` parameter containing the value of the `epc` field of the Tag represented in the `epc-decimal` format. If false, each `ECReportGroupListMember` SHALL NOT include the `rawDecimal` parameter.
`includeCount`	`Boolean`	If `includeCount` is true, the `groupCount` parameter of each generated `ECReportGroup` instance SHALL be set to an `ECReportGroupCount` instance, giving the number of Tags in the group. If false, the `groupCount` parameter in each generated `ECReportGroup` instance SHALL be set to null.
`fieldList`	`List<ECReportOutputFieldSpec>`	An ordered list of fields to include in the result. If specified and non-empty, each generated `ECReportGroupListMember` instance SHALL include a `fieldList` parameter, with contents as specified in Section 8.3.6. If empty or null, each generated `ECReportGroupListMember` SHALL NOT include the `fieldList` parameter.

Table 41. `ECReportOutputSpec` Instance

The `define` and `immediate` methods SHALL raise an `ECSpecValidationException` if any of the following are true for any `ECReportOutputSpec` instance:

- Two members of `fieldList` have the same name (after applying defaults as specified in Section 8.2.11).

- Any member of `fieldList` has a `fieldspec` parameter that is an invalid `ECFieldSpec` (see Section 8.2.12).
- All five booleans `includeEPC`, `includeTag`, `includeRawHex`, `includeRawDecimal`, and `includeCount` are false, `fieldList` is empty or omitted, and there is no vendor extension to `ECReportOutputSpec`.

8.2.11 ECReportOutputFieldSpec

An `ECReportOutputFieldSpec` specifies a Tag field to be included in an event cycle report.

ECReportOutputFieldSpec
fieldspec : ECFieldSpec
name : String // optional
includeFieldSpecInReport : Boolean // optional
<<extension point>>

The ALE implementation SHALL interpret the fields of an `ECReportOutputFieldSpec` as follows.

Field	Type	Description
`fieldspec`	`ECFieldSpec`	Specifies which field of the Tag is to be included in the report. The `fieldspec` may contain a “pattern” fieldname, in which case zero or more fields matching the pattern are read and included in the report.
`name`	`String`	(Optional) Specifies a name that is included in the corresponding `ECReportGroupListMember` instance. If empty or null, the `fieldname` parameter of the specified `fieldspec` SHALL be used as the name.
`includeFieldSpec-InReport`	`Boolean`	(Optional) If true, the corresponding `ECReportGroupListMember` instance SHALL include a copy of the specified `fieldspec`. If omitted or false, the corresponding `ECReportGroupListMember` instance SHALL NOT include a fieldspec.

Table 42. ECReportOutputFieldSpec Fields

8.2.12 ECFieldSpec

An ECFieldSpec encodes a fieldspec as defined in Section 5.4.

ECFieldSpec
fieldname : String
datatype : String
format : String
<<extension point>>

The ECFieldSpec type is used in many places within the ALE Reading API and ALE Writing API. An ALE implementation SHALL interpret an ECFieldSpec instance as follows:

Field	Type	Description
fieldname	String	Specifies the fieldname; that is, which field of the Tag to operate upon. When used in an ECReportOutputFieldSpec, may be a “pattern” fieldname that specifies zero or more fields matching the pattern.
datatype	String	(Optional) Specifies what kind of data values the field holds, and how they are encoded into Tag memory. If omitted, the ALE implementation SHALL behave as though the default datatype associated with fieldname were specified instead.
format	String	(Optional) Specifies the syntax used to present field values through the ALE interface. If omitted, the ALE implementation SHALL behave as though the default format associated with fieldname were specified instead.

Table 43. ECFieldSpec Fields

An ALE implementation SHALL consider an ECFieldSpec instance invalid if any of the following are true:

- The value of fieldname is not a valid absolute address fieldname as defined in Section 6.1.9.1, a valid variable fieldname as defined in Section 6.1.9.2, a valid variable pattern fieldname as defined in Section 6.1.9.3, the name of a built-in fieldname as defined in Section 6.1 or otherwise provided by the ALE implementation as a vendor extension, or a user-defined fieldname defined via the Tag Memory API (Section 7).

- The value of `fieldname` is a valid variable pattern fieldname as defined in Section 6.1.9.3, but the `ECFieldSpec` instance is in some context other than an `ECReportOutputFieldSpec` instance.
- The value of `datatype` is not a valid datatype for the specified `fieldname`.
- The value of `format` is not a valid format for the specified `fieldname` and specified `datatype` (or the default datatype for the specified `fieldname`, if `datatype` is omitted).

Each context where `ECFieldSpec` is used elsewhere in the specification of the Reading API and Writing API specifies what happens if an `ECFieldSpec` is invalid. (In general, an appropriate validation exception is raised.)

8.2.13 ECStatProfileName

Each valid value of `ECStatProfileName` names a statistics profile that can be included in an `ECReports`.

<< Enumerated Type>> ECStatProfileName
TagTimestamps
<<extension point>>

This specification defines one statistics profile named `TagTimestamps` which vendors MAY implement; vendors MAY also implement their own proprietary profiles.

8.2.14 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:

- The specified `specName` is an empty string or is not accepted by the implementation according to Section 4.5.
- The `logicalReaders` parameter of `ECSpec` is null, omitted, is an empty list, or contains any logical reader names that are not known to the implementation.
- The `boundarySpec` parameter of `ECSpec` is null or omitted.
- The `duration`, `stableSetInterval`, or `repeatPeriod` parameter of `ECBoundarySpec` is negative.
- The value of the `startTrigger` or `stopTrigger` parameter of `ECBoundarySpec`, or any element of the `startTriggerList` or `stopTriggerList` parameter of `ECBoundarySpec` does not conform to URI

syntax as defined by [RFC2396], or is a URI that is not supported by the ALE implementation. Note that an empty string does not conform to URI syntax as defined by [RFC2396].

- No stopping condition is specified in `ECBoundarySpec`; *i.e.*, `stopTrigger` is omitted or null, `stopTriggerList` is empty, `whenDataAvailable` is false, and neither `duration` nor `stableSetInterval` nor any vendor extension stopping condition is specified.
- The `reportSpecs` parameter of `ECSpec` is null, omitted, or empty.
- Any `ECReportSpec` instance has a `reportName` that is an empty string or that is not accepted by the implementation according to Section 4.5.
- Two `ECReportSpec` instances have identical values for their `reportName` fields.
- Any member of `includePatterns` or `excludePatterns` within `ECFilterSpec` does not conform to the `epc-tag` format's filter syntax as defined in Section 6.2.1.3.
- Two members of the `fieldList` parameter of any `ECReportOutputSpec` instance have the same name (after applying defaults as specified in Section 8.2.11).
- The `fieldspec` parameter of any `ECFilterListMember` instance is invalid according to Section 8.2.12.
- The `patList` parameter of any `ECFilterListMember` instance is empty, null, or omitted, or any element of `patList` does not conform to the syntax rules for patterns implied by the specified `fieldspec`.
- The `fieldspec` parameter of `ECGroupSpec` is invalid according to Section 8.2.12.
- The `fieldspec` parameter of `ECGroupSpec` implies a datatype and format for which no grouping pattern syntax is defined.
- Any grouping pattern within the `patternList` parameter of `ECGroupSpec` does not conform to the syntax for grouping patterns implied by the specified `fieldspec`.
- Any two grouping patterns within the `patternList` parameter of `ECGroupSpec` are not disjoint, according to the definition of disjointedness defined by the datatype and format implied by the specified `fieldspec`.
- Any member of the `fieldList` parameter within `ECReportOutputSpec` is an invalid fieldspec according to Section 8.2.12.
- Any member of the `primaryKeyFields` parameter of `ECSpec` is not a known fieldname.
- The implementation does not support the specified `primaryKeyFields` value of `ECSpec` with the specified logical readers. An implementation SHALL NOT,

however, raise the exception if `primaryKeyFields` is omitted or its value is a list consisting of the single element `epc`.

- For any `ECReportOutputSpec` instance, all five booleans `includeEPC`, `includeTag`, `includeRawHex`, `includeRawDecimal`, and `includeCount` are false, `fieldList` is empty or omitted, and there is no vendor extension to `ECReportOutputSpec`.
- Any value of `ECStatProfileName` is not recognized, or is recognized but the specified statistics report is not supported.

8.3 ECReports

`ECReports` is the output from an event cycle.

```
ECReports
specName : String
date : dateTime
ALEID : String
totalMilliseconds : long
initiationCondition : ECInitiationCondition
initiationTrigger : ECTrigger
terminationCondition : ECTerminationCondition
terminationTrigger : ECTrigger
ECSpec : ECSpec
reports : List<ECReport>
<<extension point>>
---
```

The "meat" of an `ECReports` instance is the ordered list of `ECReport` instances, each corresponding to an `ECReportSpec` instance in the event cycle's `ECSpec`, and appearing in the order corresponding to the `ECSpec`. In addition to the reports themselves, `ECReports` contains a number of "header" fields that provide useful information about the event cycle. The implementation SHALL include these fields according to the following definitions:

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.

Field	Description
`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 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.
`initiationCondition`	Indicates what kind of event caused the event cycle to initiate: the receipt of an explicit start trigger, the expiration of the repeat period, or a transition to the *requested* state when no start triggers were specified in the `ECSpec`. These correspond to the possible ways of specifying the start of an event cycle as defined in Section 8.2.1.
`initiationTrigger`	If `initiationCondition` is `TRIGGER`, the `ECTrigger` instance corresponding to the trigger that initiated the event cycle; omitted otherwise.
`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, the read field being stable for the prescribed amount of time, or the "when data available" condition becoming true. These correspond to the possible ways of specifying the end of an event cycle as defined in Section 8.2.1.
`terminationTrigger`	If `terminationCondition` is `TRIGGER`, the `ECTrigger` instance corresponding to the trigger that terminated the event cycle; omitted otherwise.
`ECSpec`	A copy of the `ECSpec` that generated this `ECReports` instance. Only included if the `ECSpec` has `includeSpecInReports` set to true.

Table 44. `ECReports` Fields

8.3.1 ECInitiationCondition

`ECInitiationCondition` is an enumerated type that describes how an event cycle was started.

<<Enumerated Type>> ECInitiationCondition
TRIGGER
REPEAT_PERIOD
REQUESTED
UNDEFINE
<<extension point>>

The ALE implementation SHALL set the `initiationCondition` field of an `ECReports` instance generated at the conclusion of an event according to the condition that caused the event cycle to start, as specified in the following table.

ECInitiationCondition	**Event causing the event cycle to start**
TRIGGER	One of the triggers specified in the `startTrigger` or `startTriggerList` parameter of `ECBoundarySpec` was received.
REPEAT_PERIOD	The `repeatPeriod` specified in the `ECBoundarySpec` expired, or the event cycle started immediately after the previous event cycle ended because neither a start trigger nor a repeat period was specified.
REQUESTED	The `ECSpec` transitioned from the unrequested state to the requested state and `startTriggerList` in `ECBoundarySpec` was empty.
UNDEFINE	Used when an outstanding `poll` call is terminated due to an `undefine` call, while the ECSpec was in the requested state (that is, before any start condition actually occurred). See Section 5.6.1.

Table 45. `ECInitiationCondition` Values

Each row of this table corresponds to one of the possible start conditions specified in Section 8.2.1.

8.3.2 ECTerminationCondition

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

<<Enumerated Type>> ECTerminationCondition
TRIGGER
DURATION
STABLE_SET
DATA_AVAILABLE
UNREQUEST
UNDEFINE
<<extension point>>

The ALE implementation SHALL set the `terminationCondition` field of an `ECReports` instance generated at the conclusion of an event cycle according to the condition that caused the event cycle to end, as specified in the following table.

ECTerminationCondition	**Event causing the event cycle to end**
TRIGGER	One of the triggers specified in `stopTriggerList` of `ECBoundarySpec` was received.
DURATION	The `duration` specified in the `ECBoundarySpec` expired.
STABLE_SET	No new Tags were read within the `stableSetInterval` specified in the `ECBoundarySpec`.
DATA_AVAILABLE	The `whenDataAvailable` parameter of the `ECSpec` was true and a Tag was read.
UNREQUEST	The `ECSpec` transitioned to the *unrequested* state. By definition, this value cannot actually appear in an `ECReports` instance sent to any client.
UNDEFINE	The `ECSpec` was removed by an `undefine` call while in the requested or active state. See Section 5.6.1.

Table 46. `ECTerminationCondition` Values

Each row of this table corresponds to one of the possible stop conditions specified in Section 8.2.1.

8.3.3 ECReport

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

```
ECReport
---
reportName : String
groups : List<ECReportGroup>
<<extension point>>
---
```

An ALE implementation SHALL construct an `ECReport` as follows:

Field	Type	Description
`reportName`	`String`	A copy of the `reportName` field from the corresponding `ECReportSpec` within the `ECSpec` that controlled this event cycle.
`groups`	`List<ECReportGroup>`	An unordered 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.

Table 47. `ECReport` Fields

8.3.4 ECReportGroup

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

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

An ALE implementation SHALL construct an `ECReportGroup` as follows:

Field	Type	Description
`groupName`	`String`	Null for the default group. For any other group, the group name as determined according to Section 8.2.9.

Field	Type	Description
groupList	ECReportGroupList	Null if the includeEPC, includeTag, includeRawHex, and includeRawDecimal fields of the corresponding ECReportOutputSpec are all false and the fieldList in the corresponding ECReportOutputSpec is empty (unless ECReportOutputSpec has vendor extensions that cause groupList to be included). Otherwise, an ECReportGroupList instance containing data read from the Tags in this group.
groupCount	ECReportGroupCount	Null if the includeCount field of the corresponding ECReportOutputSpec is false (unless ECReportOutputSpec has vendor extensions that cause groupCount to be included). Otherwise, the number of Tags in this group.

Table 48. ECReportGroup Fields

8.3.5 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.

ECReportGroupList
members : List<ECReportGroupListMember>
<<extension point>>

An ALE implementation SHALL construct an ECReportGroupList as follows:

Field	Type	Description
members	List<ECReport GroupListMember>	An unordered, possibly empty list of ECReportGroupListMember instances, one for each distinct Tag that belongs to this group. See the note below.

Table 49. ECReportGroupList Fields

Each distinct Tag included in this group SHALL have a distinct `ECReportGroupListMember` element in the `ECReportGroupList`, even if those `ECReportGroupListMember` elements would be identical due to the fields and formats selected. For example, it is possible for two different tags to have the same pure identity EPC representation; e.g., two Tags having SGTIN-96 EPC values 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. Similarly, if two Tags have the same values in one or more user defined fields, and `ECReportOutputSpec` only specified reading from those fields, the resulting `ECReportGroupList` SHALL have two `ECReportGroupListMember` elements, each having the same user fields in the `fieldList` parameter.

8.3.6 ECReportGroupListMember

Each member of the `ECReportGroupList` is an `ECReportGroupListMember` as defined below.

ECReportGroupListMember
epc : URI
tag : URI
rawHex : URI
rawDecimal : URI
fieldList : List<ECReportMemberField>
stats : List<ECTagStat>
<<extension point>>

An ALE implementation SHALL construct an `ECReportGroupListMember` from information read from a single Tag, as follows:

Field	Type	Description
epc	URI	Null, if the `includeEPC` field of the corresponding `ECReportOutputSpec` instance is false, or if accessing the `epc` field of the Tag results in a "field not found" or "operation not possible" condition. Otherwise, the value of the `epc` field of the Tag, in the `epc-pure` format as specified in Section 6.2.1.1.

Field	Type	Description
tag	URI	Null, if the includeTag field of the corresponding ECReportOutputSpec instance is false, or if accessing the epc field of the Tag results in a "field not found" or "operation not possible" condition. Otherwise, the value of the epc field of the Tag, in the epc-tag format as specified in Section 6.2.1.1.
rawHex	URI	Null, if the includeRawHex field of the corresponding ECReportOutputSpec instance is false, or if accessing the epc field of the Tag results in a "field not found" or "operation not possible" condition. Otherwise, the value of the epc field of the Tag, in the epc-hex format as specified in Section 6.2.1.1.
rawDecimal	URI	Null, if the includeRawDecimal field of the corresponding ECReportOutputSpec instance is false, or if accessing the epc field of the Tag results in a "field not found" or "operation not possible" condition. Otherwise, the value of the epc field of the Tag, in the epc-decimal format as specified in Section 6.2.1.1.
fieldList	List<ECReportMemberField>	Null, if the fieldList parameter of the corresponding ECReportOutputSpec is empty, omitted, or null. Otherwise, contains zero or more ECReportMemberField instances for each fieldspec listed in the fieldList parameter of the corresponding ECReportOutputSpec, in the corresponding order. If a fieldspec specified a pattern fieldname, then zero or more ECReportMemberField instances may be present. Otherwise, exactly one ECReportMemberField instance is present.
stats	List< ECTagStat>	Null, if the statProfileNames parameter of the corresponding ECReportSpec is empty, omitted, or null. Otherwise, contains an ECTagStat for each statistics profile named in the statProfileNames parameter of the corresponding ECReportSpec, in the corresponding order.

Table 50. `ECReportGroupListMember` Fields

8.3.7 ECReportMemberField

Each `ECReportMemberField` within the `fieldList` of an `ECReportGroupListMember` gives the value read from a single field of a single Tag.

`ECReportMemberField`
`name : String`
`value : String // optional`
`fieldspec : ECFieldSpec // optional`
`<<extension point>>`
`---`

An ALE implementation SHALL construct an `ECReportMemberField` as follows:

Field	Type	Description
`name`	`String`	The name specified in the corresponding `ECReportOutputFieldSpec` that generated this `ECReportMemberField` instance in this report, either explicitly or defaulted to the fieldname as specified in Section 8.2.11. If the name is defaulted to the fieldname, and the fieldname specified in the `ECReportOutputFieldSpec` was a pattern fieldname, then the value of the "name" parameter SHALL be the name of the specific field that matched the pattern.
`value`	`String`	(Optional) The value read from the field of the Tag. This value SHALL conform to the syntax implied by the `format` parameter of `fieldspec`. If the attempt to read the field value of the Tag caused a "field not found" or "operation not possible" condition, the `value` parameter SHALL be omitted.

Field	Type	Description
fieldspec	ECFieldSpec	(Optional) If the includeFieldSpecInReport parameter of the corresponding ECReportOutputFieldSpec that generated this ECReportMemberField instance in this report was set to true, this fieldspec parameter SHALL contain a copy of the corresponding ECFieldSpec instance in the ECReportOutputFieldSpec. If the datatype or format parameters were omitted in the original ECFieldSpec, in this copy those fields SHALL contain the default datatype or format that were used. Omitted if the includeFieldSpecInReport parameter of the corresponding ECReportOutputFieldSpec that generated this ECReportMemberField instance in this report was omitted or set to false.

Table 51. ECReportMemberField Fields

8.3.8 ECReportGroupCount

An ECReportGroupCount is included in an ECReportGroup when the includeCount field of the corresponding ECReportOutputSpec is true.

```
ECReportGroupCount
count : Integer
<<extension point>>
---
```

An ALE implementation SHALL construct an ECReportGroupCount as follows:

Field	Type	Description
count	Integer	The number of distinct Tags that are part of this group.

Table 52. ECReportGroupCount Fields

8.3.9 ECTagStat

An ECTagStat provides additional, implementation-defined information about each "sighting" of a Tag, that is, each time a Tag is acquired by one of the Readers participating in the event cycle.

ECTagStat
profile : ECStatProfileName
statBlocks : List<ECReaderStat>

An ALE implementation SHALL construct an ECTagStat as follows:

Field	Type	Description
profile	ECStatProfileName	The name of the statistics profile that governed the generation of this ECTagStat instance.
statBlocks	List<ECReaderStat>	An unordered list containing an ECReaderStat instance for each Reader that sighted this Tag.

Table 53. ECTagStat Fields

8.3.10 ECReaderStat

An ECReaderStat contains information about sightings of a Tag by a particular Reader. An ALE implementation MAY use a subclass of this type to provide information about a Reader's interaction with a Tag that is not specific to a particular sighting. For example, a subclass of this type might provide timestamps for the first and last time the Tag was sighted by the Reader, or the total number of sightings of the Tag by that Reader.

ECReaderStat
readerName : String
sightings : List<ECSightingStat>

An ALE implementation SHALL construct an ECReaderStat as follows:

Field	Type	Description
readerName	String	The name of the logical Reader whose sightings are reported in this ECReaderStat. This name may, at the implementer's discretion, refer to either a logical reader name as named in the defining ECSpec or one of the underlying component readers that contribute to a named logical reader. Implementers SHOULD document for each statistics profile which of the names are used (or both).
sightings	List<ECSightingStat>	(Optional) An unordered list containing information pertaining to one sighting of the Tag by the Reader named in readerName.

Table 54. ECReaderStat Fields

Note that ECReaderStat is used both by the Reading API and the Writing API. Unless otherwise noted, the interpretation of an ECReaderStat instance is the same in both APIs.

8.3.11 ECSightingStat

An ECSightingStat contains information about a single sighting of a Tag by a particular Reader. An ALE implementation MAY use a subclass of this type to provide information about a single sighting of a Tag. For example, a subclass of this type might provide the timestamp for this sighting, the received signal strength (for an RFID Tag), etc.

Note that ECSightingStat is used both by the Reading API and the Writing API. Unless otherwise noted, the interpretation of an ECSightingStat instance is the same in both APIs.

8.3.12 ECTagTimestampStat

ECTagTimestampStat is a subclass of ECTagStat. An ALE implementation SHALL include one ECTagTimestampStat in an ECReportGroupListMember if the TagTimestamps statistics profile was included in the corresponding ECReportSpec and the implementation chooses to implement the TagTimestamps statistics profile. ECTagTimestampStat includes all of the fields in ECTagStat, plus the following additional fields:

ECTagTimestampStat
firstSightingTime: dateTime
lastSightingTime: dateTime

An ALE implementation SHALL construct an ECTagTimestampStat as follows:

Field	Type	Description
profile	ECStatProfileName	This field SHALL contain the TagTimestamps value of ECStatProfileName.
statBlocks	List<ECReaderStat>	This field SHALL contain an empty list (i.e. a list that contains 0 items).
firstSightingTime	dateTime	If the ECReportSetSpec for this report is DELETIONS then this field MAY be present. If present, it SHALL contain the first time that the tag was seen during the previous event cycle. If the ECReportSetSpec for this report is CURRENT or ADDITIONS then this field SHALL contain the first time within this event cycle that the tag was seen by any reader contributing to this event cycle.
lastSightingTime	dateTime	If the ECReportSetSpec for this report is DELETIONS then this field MAY be present. If present, it SHALL contain the last time that the tag was seen during the previous event cycle. If the ECReportSetSpec for this report is CURRENT or ADDITIONS then this field SHALL contain the last time within this event cycle that the tag was seen by any reader contributing to this event cycle.

Table 55. ECTagTimestampStat Fields

Implementations MAY choose to use any clock that they wish to measure firstSightingTime and lastSightingTime, but they SHALL correct for any differences in clocks such that those time stamps are brought into synchronization with the date field of ECReports. For bindings in which time is represented textually, an ISO-8601 compliant representation SHOULD be used.

8.4 ALECallback Interface

The `ALECallback` interface is the path by which an ALE implementation delivers asynchronous results from event cycles to subscribers.

```
<<interface>>
ALECallback
---
callbackResults(reports : ECReports) : void
```

Referring to the state transition tables in Section 5.6.1, whenever a transition specifies that "reports are delivered to subscribers" the ALE implementation SHALL attempt to deliver the results to each subscriber by invoking the `callbackResults` method of the `ALECallback` interface once for each subscriber, passing the `ECReports` for the event cycle as specified above, and using the binding and addressing information specified by the notification URI for that subscriber as specified in the `subscribe` call. All subscribers receive an identical `ECReports` instance.

Explanation (non-normative): The `ALECallback` interface is defined very simply, to allow for a wide variety of possible implementations. A binding of the `ALECallback` interface may not be a request-response style RPC mechanism at all, but may instead just be a one-way message transport, where the message payload is the `ECReports` instance. Indeed, this is true of all of the standardized bindings of this interface described in Part II [ALE1.1Part2].

9 ALE Writing API

This section defines normatively the ALE Writing API. The external interface is defined by the `ALECC` interface (Sections 9.1, 9.5, 9.6, and 9.7). This interface makes use of a number of complex data types that are documented in the sections following Section 9.1. The specification of the Writing API follows the general rules given in Section 4.

Through the `ALECC` interface defined in Section 9.1, clients may define and manage command cycle specifications (CCSpecs), operate upon Tags on-demand by activating CCSpecs synchronously, and enter standing requests (subscriptions) for CCSpecs to be activated asynchronously. Results from standing requests are delivered through the `ALECCCallback` interface, specified in Section 9.8.

Implementations MAY expose the `ALECC` interface of the ALE Writing API interface via a wire protocol, or via a direct API in which clients call directly into code that implements the API. Likewise, implementations MAY implement the `ALECCCallback` interface via a wire protocol or via a direct API in which clients receive asynchronous results through a direct callback. This Part I of the ALE 1.1 specification does not define the concrete wire protocol or programming language-specific API, but instead only provides an abstract specification of the interfaces using UML. Part II of the specification [ALE1.1Part2] specifies XML-based wire protocol bindings of the interfaces, including an XSD schema for the API data types, a WS-I compliant WSDL definition of a SOAP binding of the ALECC interface, and several XML-based bindings of the ALECCCallback interface. Implementations MAY provide additional bindings of the API, including bindings to particular programming languages.

9.1 ALECC Class

```
<<interface>>
ALECC
---
define(specName : String, spec : CCSpec) : void
undefine(specName : String) : void
getCCSpec(specName : String) : CCSpec
getCCSpecNames() : List<String>   // returns a list of specNames as strings
subscribe(specName : String, notificationURI : String) : void
unsubscribe(specName : String, notificationURI : String) : void
poll(specName : String, params : CCParameterList) : CCReports
immediate(spec : CCSpec) : CCReports
getSubscribers(specName : String) : List<String> // of notification URIs
getStandardVersion() : String
getVendorVersion() : String
<<extension point>>
```

An ALE implementation SHALL implement the above methods of the ALE Writing API as specified in the following table:

Method	Argument/ Result	Type	Description
define	specName	String	Creates a new CCSpec having the name specName, according to spec. The lifecycle of the new CCSpec SHALL be subject to the provisions of Section 5.6.1.
	spec	CCSpec	
	[result]	Void	
undefine	specName	String	Removes the CCSpec named

Method	Argument/ Result	Type	Description
	[result]	Void	specName that was previously created by the define method. The effect SHALL be as specified in Section 5.6.1.
getCCSpec	specName	String	Returns the CCSpec that was provided when the CCSpec named specName was created by the define method. The result SHALL be equivalent to the CCSpec that was provided to the define method, but NEED NOT be identical. "Equivalent" means that the returned CCSpec has exactly the same meaning as the original CCSpec when interpreted according to this specification.
	[result]	CCSpec	
getCCSpecNames	[result]	List< String>	Returns an unordered list of the names of all CCSpecs that are visible to the caller. The order of this list is implementation-dependent.
subscribe	specName	String	Adds a subscriber having the specified notificationURI to the set of current subscribers of the CCSpec named specName. The effect SHALL be as specified in Section 5.6.1. The notificationURI parameter both identifies a specific binding of the ALECCCallback interface and specifies addressing information meaningful to that binding. See Part II.
	notifi-cationURI	String	
	[result]	void	
unsubscribe	specName	String	Removes a subscriber having the specified notificationURI from the set of current
	notify-cationURI	String	

Method	Argument/ Result	Type	Description
	[result]	`void`	subscribers of the CCSpec named `specName`. The effect SHALL be as specified in Section 5.6.1.
`poll`	`specName`	`String`	Requests an activation of the CCSpec named `specName`, returning the results from the next event cycle to complete, as specified in Section 5.6.1. Within this activation, `params` provides the values for parameters referred to in `CCOpSpec` instances. See also the text at the end of Section 5.6.1. The ALE implementation MAY provide a means to abort an outstanding `poll` call, by explicit client action, by timeout, or by some other means. If such a means is provided, the effect on the CCSpec lifecycle of aborting the `poll` call SHALL be as specified in Section 5.6.1.
	`params`	`CCPara-meterList`	
	[result]	`CCReports`	
`immediate`	`spec`	`CCSpec`	Creates an unnamed CCSpec according to spec, and immediately requests its activation. The behavior SHALL be, as specified in Section 5.6.2. The ALE implementation MAY provide a means to abort an outstanding `immediate` call, by explicit client action, by timeout, or by some other means. If such a means is provided, the effect on the CCSpec lifecycle of aborting the `immediate` call SHALL be as specified in Section 5.6.2.
	[result]	`CCReports`	

Method	Argument/ Result	Type	Description
`getSubscribers`	`specName`	`String`	Returns an unordered, possibly empty list of the notification URIs corresponding to each of the current subscribers for the CCSpec named `specName`.
	[result]	`List<String>`	
`getStandardVersion`	[result]	`String`	Returns a string that identifies what version of the specification this implementation of the ALE Writing API complies with as specified in Section 4.3.
`getVendorVersion`	[result]	`String`	Returns a string that identifies what vendor extensions of the ALE Writing API this implementation provides as specified in Section 4.3.

Table 56. ALECC Interface Methods

The primary data types associated with the ALE Writing API are the `CCSpec`, which specifies how a command cycle is to be carried out, and the `CCReports`, which contains one or more reports generated from one activation of a `CCSpec`. `CCReports` instances are both returned from the `poll` and `immediate` methods, and also sent to subscribers when `CCSpecs` are subscribed to using the `subscribe` method. The next two sections, Section 9.3 and Section 9.4, specify the `CCSpec` and `CCReports` data types in full detail.

9.1.1 Error Conditions

Methods of the ALE Writing 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. If the Writing API implementation is associated with an implementation of the Access Control API (Section 11), the Writing API implementation SHALL raise this exception if the client was not granted access rights to the called method as specified in Section 11. Other, implementation-specific circumstances may cause this exception; these are outside the scope of this specification.
`DuplicateNameException`	The specified CCSpec name already exists. Note that the existence of an ECSpec having the same name does *not* cause this exception; ECSpecs and CCSpecs are in different namespaces.
`CCSpecValidationException`	The specified CCSpec is invalid. The complete list of rules for generating this exception is specified in Section 9.3.10.
`InvalidURIException`	The URI specified for a subscriber does not conform to URI syntax as specified in [RFC2396], does not name a binding of the `ALECCCallback` interface recognized by the implementation, or violates syntax or other rules imposed by a particular binding.
`NoSuchNameException`	The specified CCSpec name does not exist.
`NoSuchSubscriberException`	The specified subscriber does not exist.
`DuplicateSubscriptionException`	The specified CCSpec name and subscriber URI is identical to a previous subscription that was created and not yet unsubscribed.

Exception Name	Meaning
`ParameterException`	The specified `params` parameter of the `poll` method was invalid for any of the following reasons: • Two or more `CCParameterListEntry` instances have the same `name`. • A `CCSpec` refers to a parameter, but the `params` parameter to `poll` lacks an entry for that parameter name. • The `value` of a `CCParameterListEntry` is not valid syntax for the datatype and format implied by the `fieldspec` of a `CCOpDataSpec` that refers to that parameter.
`ParameterForbiddenException`	The CCSpec referred to by a `subscribe` or `immediate` operation includes a `CCOpDataSpec` of type `PARAM`.
`ImplementationException`	A generic exception raised 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.

Table 57. Exceptions in the ALECC Interface

The exceptions that may be raised by each ALE method are indicated in the table below. An ALE implementation SHALL raise the appropriate exception listed below when the corresponding condition described above occurs. If more than one exception condition applies to a given method call, the ALE implementation may raise any of the exceptions that applies.

ALE Method	Exceptions
define	DuplicateNameException CCSpecValidationException SecurityException ImplementationException
undefine	NoSuchNameException SecurityException ImplementationException
getCCSpec	NoSuchNameException SecurityException ImplementationException
getCCSpecNames	SecurityException ImplementationException
subscribe	NoSuchNameException InvalidURIException DuplicateSubscriptionException ParameterForbiddenException SecurityException ImplementationException
unsubscribe	NoSuchNameException NoSuchSubscriberException InvalidURIException SecurityException ImplementationException
poll	NoSuchNameException ParameterException SecurityException ImplementationException
immediate	CCSpecValidationException ParameterForbiddenException SecurityException ImplementationException
getSubscribers	NoSuchNameException SecurityException ImplementationException
getStandardVersion	ImplementationException
getVendorVersion	ImplementationException

Table 58. Exceptions Raised for each ALECC Interface Method

9.2 CCParameterList

A CCParameterList is an unordered list of name/value pairs, each specifying a parameter name and a corresponding parameter value. Parameter values are string data that provide specific values to be used in tag commands. See Sections 9.3.4 and 9.3.5.

CCParameterList
`entries : List<CCParameterListEntry>`
`---`

9.2.1 CCParameterListEntry

A `CCParameterListEntry` is a single name/value pair.

CCParameterListEntry
`name : String`
`value : String`
`---`

9.3 CCSpec

A CCSpec is a complex type that describes a command cycle. A command cycle is an interval of time during which Tags are operated upon.

A CCSpec contains (a) one or more logical reader names; (b) a boundary specification (`CCBoundarySpec`) that identifies an interval of time; (c) one or more command specifications (`CCCmdSpec`) that specify operations to be performed on a population of Tags visible to the specified logical readers during the specified interval of time. The command specifications also imply what information is included in a report generated from each command cycle generated from this CCSpec.

CCSpec
`logicalReaders : List<String> // List of logical reader names`
`boundarySpec : CCBoundarySpec`
`cmdSpecs : List<CCCmdSpec>`
`includeSpecInReports : Boolean`
`<<extension point>>`
`---`

The ALE implementation SHALL interpret the fields of a `CCSpec` as follows.

Field	Type	Description
logicalReaders	List<String>	An unordered list that specifies one or more logical readers that are used to reach tags.
boundarySpec	CCBoundarySpec	Specifies the starting and stopping conditions for command cycles. See Section 9.3.1.
cmdSpecs	List<CCCmdSpec>	An ordered list that specifies one or more sequences of commands to apply to Tags. See Section 9.3.2.
includeSpecInReports	Boolean	If true, specifies that each CCReports instance generated from this CCSpec SHALL include a copy of the CCSpec. If false, each CCReports instance SHALL NOT include a copy of the CCSpec.

Table 59. CCSpec Fields

The define and immediate methods SHALL raise a CCSpecValidationException if any of the following are true for a CCSpec instance:

- The logicalReaders parameter is null, omitted, is an empty list, or contains any logical reader names that are not known to the implementation.
- The boundarySpec parameter is null or omitted, or the specified boundarySpec leads to a CCSpecValidationException as specified in Section 9.3.1.
- The cmdSpecs parameter is null, omitted, empty, or any of the members of cmdSpecs leads to a CCSpecValidationException as specified in Section 9.3.2.

9.3.1 CCBoundarySpec

A CCBoundarySpec specifies how the beginning and end of command cycles are to be determined.

CCBoundarySpec
startTriggerList : List<ECTrigger>
repeatPeriod : ECTime
stopTriggerList : List<ECTrigger>
duration : ECTime
noNewTagsInterval : ECTime
tagsProcessedCount : Integer
afterError : Boolean
<<extension point>>

The ALE implementation SHALL interpret the fields of a `CCBoundarySpec` as follows.

Field	Type	Description
`startTriggerList`	`List<ECTrigger>`	(Optional) An unordered list that specifies zero or more triggers that may start a new command cycle for this `CCSpec`.
`repeatPeriod`	`ECTime`	(Optional) Specifies an interval of time for starting a new command cycle for this `CCSpec`, relative to the start of the previous command cycle.
`stopTriggerList`	`List<ECTrigger>`	(Optional) An unordered list that specifies zero or more triggers that may stop a command cycle for this `CCSpec`.
`duration`	`ECTime`	(Optional) Specifies an interval of time for stopping a command cycle for this `CCSpec`, relative to the start of the command cycle. If omitted or equal to zero, has no effect on the stopping of the command cycle.

Field	Type	Description
`noNewTagsInterval`	`ECTime`	(Optional) Specifies that a command cycle may be stopped if no new tags are encountered within the specified interval. If omitted or equal to zero, has no effect on the stopping of the command cycle.
`tagsProcessedCount`	`Integer`	(Optional) Specifies that a command cycle may be stopped after the specified number of Tags have been processed. If omitted or equal to zero, has no effect on the stopping of the command cycle.
`afterError`	`Boolean`	(Optional) If true, specifies that a command cycle may be stopped when an error is encountered during Tag processing. If omitted or false, has no effect on the stopping of the event cycle.

Table 60. `CCBoundarySpec` Fields

The `define` and `immediate` methods SHALL raise a `CCSpecValidationException` if any of the following are true for a `CCBoundarySpec` instance:

- A negative number is specified for any of the `ECTime` values `duration`, `repeatPeriod`, or `noNewTagsInterval`.
- Any element of `startTriggerList` or `stopTriggerList` does not conform to URI syntax as defined by [RFC2396], or is a URI that is not supported by the ALE implementation. Note that an empty string does not conform to URI syntax as defined by [RFC2396].
- A negative number is specified for `tagsProcessedCount`.
- No stopping condition apart from `afterError` is specified; *i.e.*, `stopTriggerList` is empty, `duration` is zero or omitted, `noNewTagsInterval` is zero or omitted, `tagsProcsssedCount` is zero or omitted, and no vendor extension stopping condition is specified.

In the description below, the phrase "if specified" used in reference to `repeatPeriod`, `duration`, `noNewTagsInterval`, or `tagsProcessedCount` means that the parameter is specified and is a positive (non-zero) number.

The boundarySpec parameter of CCSpec (of type CCBoundarySpec) specifies starting and stopping conditions as referred to in the CCSpec lifecycle specified in Sections 5.6.1 and 5.6.2. Within that description, "arrival of a start trigger" means that the ALE implementation receives any of the triggers specified in startTriggerList, and "repeat period" means the value of the repeatPeriod parameter, if specified. The phrase "a stopping condition has occurred" means the first of the following to occur:

- The duration, when specified, expires (measured from the start of the command cycle).
- When the noNewTagsInterval is specified, no new Tags are encountered by any Reader for the specified interval. In this context, "new" is to be interpreted collectively among Readers contributing to this command cycle.
- Any one of the stop triggers specified in stopTriggerList is received.
- The tagsProcessedCount parameter is specified, and that many Tags have been processed. If several matching Tags are processed in a single reader cycle, the implementation MAY terminate the command cycle after processing all of those Tags (that is, the implementation does not have to count Tags at any finer granularity than a reader cycle). Note that the only tags that count towards tagsProcessedCount are those that match the filtering conditions of at least one CCCmdSpec.
- The afterError parameter is true, and processing of a CCOpSpec for a Tag has resulted in an error. If several Tags are processed in a single reader cycle and only one results in an error, the implementation MAY terminate the command cycle after processing all of those Tags (that is, the implementation does not have to detect errors at any finer granularity than a reader cycle).

9.3.2 CCCmdSpec

A CCCmdSpec includes (a) a filter specification (CCFilterSpec) that has inclusive/exclusive filters to select a population of tags; (b) an ordered list of one or more operation specifications (CCOpSpec), each of which describes a single operation to be performed on a tag. During a command cycle, the ALE implementation attempts to carry out the commands specified by the operation specifications on each of the tags selected by the filter specification.

```
CCCmdSpec
name : String
filterSpec : CCFilterSpec
opSpecs : List<CCOpSpec>
reportIfEmpty : Boolean
statProfileNames : List<CCStatProfileName>
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of an `CCCmdSpec` as follows.

Field	Type	Description
`name`	`String`	Specifies a name for this `CCCmdSpec`. The ALE implementation SHALL copy this name into the `CCReport` instance generated from this `CCCmdSpec`.
`filterSpec`	`CCFilterSpec`	Specifies which Tags are to be processed according to this `CCCmdSpec`.
`opSpecs`	`List<CCOpSpec>`	An ordered list of `CCOpSpec` instances, each specifying an operation to be carried out on a Tag. The ALE implementation SHALL process each Tag that matches `filterSpec` acquired during a command cycle in a manner equivalent to carrying out the operations specified in `opSpecs` in the order specified. The ALE implementation MAY actually carry out operations in any order it wishes, so long as the net effect is identical to carrying them out in the order specified. For example, if two operations specify overlapping writes to user memory, the implementation may merge these into one interaction with a reader if the net result is the same.

Field	Type	Description
reportIfEmpty	Boolean	Specifies whether to omit the CCReport instance if the set of Tags matching the filterSpec parameter is empty.
statProfileNames	List<CCStat ProfileName>	An ordered list that specifies zero or more statistics profiles that govern what statistics are to be included in the report, as specified in Section 9.3.9.

Table 61. CCCmdSpec Fields

The define and immediate methods SHALL raise a CCSpecValidationException if any of the following are true for a CCCmdSpec instance:

- The specified name is an empty string or is not accepted by the implementation according to Section 4.5.
- The specified name is a duplicate of another CCCmdSpec name in the same CCSpec.
- The specified filterSpec leads to a CCSpecValidationException as specified in Section 9.3.3.
- The specified opSpecs leads to a CCSpecValidationException as specified in Section 9.3.4.
- Any element of statProfileNames is not the name of a known statistics profile.

A CCReports instance SHALL include an CCReport instance corresponding to each CCCmdSpec in the governing CCSpec, in the same order specified in the CCSpec, except that a CCReport instance SHALL be omitted under the following circumstance:

- If a CCReportSpec has reportIfEmpty set to false, then the corresponding CCReport instance SHALL be omitted from the CCReports for this command cycle if the final, filtered set of Tags is empty (i.e., if there are no Tags to operate upon).

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

Explanation (non-normative): The `name` *parameter is an arbitrary string that is copied to the* `CCReport` *instance created when this command cycle completes. The purpose of the* `name` *parameter is so that clients can distinguish which of the* `CCReport` *instances that it receives corresponds to which* `CCCmdSpec` *instance contained in the original* `CCSpec`*. This is especially useful in cases where fewer reports are delivered than there were* `CCCmdSpec` *instances in the* `CCSpec`*, because a* `reportIfEmpty=false` *setting suppressed the generation of some reports.*

9.3.3 CCFilterSpec

A `CCFilterSpec` specifies what Tags are to be processed by a `CCCmdSpec`.

CCFilterSpec
filterList : List<ECFilterListMember>
<<extension point>>

The ALE implementation SHALL interpret the fields of a `CCFilterSpec` as follows.

Field	Type	Description
`filterList`	`List<ECFilterListMember>`	Specifies an unordered list of filters, as specified below.

Table 62. `CCFilterSpec` Fields

The `define` and `immediate` methods SHALL raise a `CCSpecValidationException` if any of the following are true for a `CCFilterSpec` instance:

- Any element of `filterList` is leads to a `CCSpecValidationException` as specified in Section 8.2.8.

The `CCFilterSpec` implements a flexible filtering scheme based on a list of `ECFilterListMember` instances (`ECFilterListMember` is shared with the ALE Reading API, and is specified in Section 8.2.8). Each `ECFilterListMember` instance defines a test to be applied to fields of a Tag to determine if the Tag should be processed according to the containing `CCCmdSpec`. A Tag SHALL be subject to the operations specified in the `CCCmdSpec` if it passes the test specified by *every* `ECFilterListMember` in `filterList`, as defined in Sections 8.2.7 and 8.2.8.

If accessing a field specified by any element of `filterList` causes a “field not found” or “operation not possible” condition, that Tag SHALL not be processed as part of this `CCCmdSpec`.

9.3.4 CCOpSpec

Each `CCOpSpec` specifies an operation to perform on a Tag, such as reading a field, writing a field, or other Tag operation. Each `CCOpSpec` has an operation type that specifies which operation to perform. Operations that apply to a specific field of memory include a fieldspec that indicates which field is involved. Operations that require input data (such as writing to a field of a Tag) include a `CCOpDataSpec` to specify the input data. See Section 5.4 for an explanation of how different kinds of operations apply to different types of fields in the Tag data model.

```
CCOpSpec
opType : CCOpType
fieldspec : ECFieldSpec
dataSpec : CCOpDataSpec
opName : String
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of a `CCOpSpec` as follows.

Field	Type	Description
`opType`	`CCOpType`	Specifies the operation to be performed.
`fieldspec`	`ECFieldSpec`	(Conditional) If `opType` specifies an operation that requires a `fieldspec`, this parameter must be included to specify what field is to be operated upon and the datatype and format to be used. If `opType` specifies an operation that does not require a `fieldspec`, this parameter must be omitted.

Field	Type	Description
`dataSpec`	`CCOpDataSpec`	(Conditional) If `opType` specifies that requires input data, this parameter must be included to specify the input data. If `opType` specifies an operation that does not require input data, this parameter must be omitted.
`opName`	`String`	(Optional) A name for this operation within the `CCCmdSpec`. If specified, the value is copied into the `opName` parameter of the corresponding `CCOpReport` instance. If omitted, the `opName` parameter of the corresponding `CCOpReport` instance will be omitted as well.

Table 63. `CCOpSpec` Fields

The `define` and `immediate` methods SHALL raise a `CCSpecValidationException` if any of the following are true for a `CCOpSpec` instance:

- The specified `opType` value is not one of the standard `opType` values specified in Section 9.3.5, or an implementation-specific value known to the ALE implementation.
- The specified `opType` requires a `fieldspec`, and `fieldspec` is null or omitted.
- The specified `opType` does not require a `fieldspec`, and `fieldspec` is specified.
- The specified `fieldspec` is invalid according to Section 8.2.12.
- The specified `opType` requires a `dataSpec`, and `dataSpec` is null or omitted.
- The specified `opType` does not require a `dataSpec`, and `dataSpec` is specified.
- The specified `dataSpec` is invalid according to Section 9.3.6.
- The specified `dataSpec` specifies a value that is invalid for the specified operation, as specified in Section 9.3.6.

- When `opName` is specified, the specified `opName` is the same as an `opName` of another `CCOpSpec` within the same `CCCmdSpec` instance.

9.3.5 CCOpType

`CCOpType` is an enumerated type denoting what type of operation is represented by the `CCOpSpec`.

<<Enumerated Type>> CCOpType
READ
CHECK
INITIALIZE
ADD
WRITE
DELETE
PASSWORD
KILL
LOCK
<<extension point>>

The following table describes each value of `CCOpType`, and the interpretation of `fieldspec` and `dataSpec` within `CCOpSpec` when that `CCOpType` value is specified. Unless otherwise noted, any type of `dataSpec` may be specified.

`CCOpType` **Value**	**Description**	`fieldspec`	`dataSpec`
`READ`	Read from memory	The field to read	[Must be omitted]
`CHECK`	Check memory bank contents for consistency.	The memory bank to be checked: one of the values specified in Section 9.3.5.1	A LITERAL `dataSpec` whose value specifies the encoding of the memory bank. See Section 9.3.5.1

CCOpType **Value**	**Description**	fieldspec	dataSpec
INITIALIZE	Initialize the state of a memory so that variable fields may be used	The memory bank to initialize: one of the values specified in Section 9.3.5.2	A LITERAL dataSpec whose value specifies additional information that guides the initialization. See Section 9.3.5.2
ADD	Add the specified field to the Tag's memory, initialized to the specified value. For a fixed field, this operation is equivalent to WRITE.	The field to add	The value to write into the specified field
WRITE	Write a new value to an existing field.	The field to write	The value to write into the specified field
DELETE	Delete the specified field from memory. For a fixed field, this operation is equivalent to WRITE with a value of zero.	The field to delete	[Must be omitted]
PASSWORD	Provide a password to enable subsequent commands; for Gen2 Tags, this transitions the tag to the "secured" state.	[Must be omitted. The datatype for the input is uint and the format is hex]	The access password
KILL	Kill a tag; for Gen2 Tags this means to use the Gen2 "kill" command.	[Must be omitted. The datatype for the input is uint and the format is hex]	The kill password

CCOpType **Value**	**Description**	fieldspec	dataSpec
LOCK	Sets access permissions for a memory field	The field whose permissions are to be set	A LITERAL dataSpec whose value specifies the lock action to be performed. See enumeration values for allowed lock actions in CCLockOperation.

Table 64. CCOpType Values

9.3.5.1 Values for the CHECK Operation

An ALE implementation SHALL recognize the values defined in the following sub-sections as valid operands for the CHECK CCOpSpecType.

9.3.5.1.1 EPC/UII Memory Bank CHECK Operation

When the fieldspec is epcBank (EPC/UII memory bank), CHECK dataSpec values of the following forms SHALL be recognized:

urn:epcglobal:ale:check:iso15962

When interacting with a Gen2 Tag, an ALE implementation SHALL check the EPC/UII memory bank (Bank 01) of the Tag as follows. A CCOpStatus of MEMORY_CHECK_ERROR SHALL be indicated if any of the following are true:

- The toggle bit (bit 17h) is equal to zero.
- The AFI bits (bits 18h-1Fh) do not contain an ISO 15962 Application Family Identifier (AFI) that is recognized by the implementation.
- The memory bank does not contain an ISO 15962 Data Storage Format Identifier (DSFID) that is recognized by the implementation.
- The remaining contents of the memory bank are not valid according to ISO 15962 [ISO15962].
- The remaining contents of the memory bank include two or more data sets having the same object identifier (OID).

9.3.5.1.2 User Memory Bank CHECK Operation

When the fieldspec is userBank (EPC/UII memory bank), CHECK dataSpec values of the following forms SHALL be recognized:

urn:epcglobal:ale:check:iso15962

When interacting with a Gen2 Tag, an ALE implementation SHALL check the User memory bank (Bank 11) of the Tag as follows. A `CCOpStatus` of `MEMORY_CHECK_ERROR` SHALL be indicated if any of the following are true:

- The memory bank does not contain an ISO 15962 Data Storage Format Identifier (DSFID) that is recognized by the implementation.
- The remaining contents of the memory bank are not valid according to ISO 15962 [ISO15962].
- The remaining contents of the memory bank include two or more data sets having the same object identifier (OID).

9.3.5.2 Values for the INITIALIZE Operation

An ALE implementation SHALL recognize the values defined in the following sub-sections as valid operands for the `INITIALIZE` `CCOpSpecType`.

An ALE implementation SHALL raise a `CCSpecValidationException` if the combination of fieldspec and value for the `INITIALIZE` `CCOpSpecType` are not recognized.

9.3.5.2.1 EPC/UII Memory Bank INITIALIZE Operation

When the fieldspec is `epcBank` (EPC/UII memory bank), INITIALIZE `dataSpec` values of the following forms SHALL be recognized:

`urn:epcglobal:ale:init:iso15962:x`*AA*`[.x`*DD*`][.force]`

where *AA* denotes two hexadecimal digits and *DD* denotes two or more hexadecimal digits. When interacting with a Gen2 Tag, an ALE implementation SHALL initialize the EPC/UII memory bank (Bank 01) of the Tag as follows:

Write a one into bit 17h, write the value *AA* into bits 18h-1Fh, write the value *DD* beginning at bit 20h (the number of bits so written being four times the number of characters in *DD*), followed by eight zero bits (note: the eight zero bits indicate that there are no ISO data sets in the EPC/UII memory bank). Subsequent operations on the Tag will interpret *AA* as the ISO 15962 Application Family Identifier (AFI), and *DD* as the ISO 15962 Data Storage Format Identifier (DSFID). If `x`*DD* is omitted, the ALE implementation SHALL supply a default value for *DD*. The ALE implementation MAY examine subsequent commands in the `CCCmdSpec` to make an appropriate choice, based on the particular OID or OIDs to be written to the tag, possibly combined with other ALE settings that could for example convey the application's desire to use an external Directory structure (or other special features that a DSFID can indicate) with the tag.

If the optional `.force` is not present in the `dataSpec` value, then the ALE implementation SHALL omit all initialization steps as described above if the prior contents of the bits 17h is a one, and the prior contents of bits 18h through 27h are non-zero.

Explanation (non-normative); In other words, if `.force` is omitted, then the tag is initialized only if it was not previously initialized to a valid ISO 15962 memory state (though not necessarily having the specified AFI and DSFID). Previously existing data sets are preserved in that case. If the tag was not previously initialized, or if `.force` is specified, then the memory is always initialized to a state containing no data sets.

When interacting with a Gen1 Tag, the implementation SHALL raise an "operation not possible" condition. When interacting with any other type of Tag, the interpretation of INITIALIZE on the EPC/UII bank is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

9.3.5.2.2 User Memory Bank INITIALIZE Operation

When the fieldspec is `userBank` (User memory bank), INITIALIZE `dataSpec` values of the following form SHALL be recognized:

`urn:epcglobal:ale:init:iso15962:[x`***`DD`***`][.force]`

where ***DD*** denotes two or more hexadecimal digits. When interacting with a Gen2 Tag, an ALE implementation SHALL initialize the User memory bank (Bank 11) of the Tag as follows:

Write the value ***DD*** beginning at bit 00h (the number of bits so written being four times the number of characters in ***DD***), followed by eight zero bits (note: the eight zero bits indicate that there are no ISO data sets in the User memory bank). Subsequent operations on the Tag will interpret ***DD*** as the ISO 15962 Data Storage Format Identifier (DSFID). If `x`***`DD`*** is omitted, the ALE implementation SHALL supply a default value for ***DD***. The ALE implementation MAY examine subsequent commands in the `CCCmdSpec` to make an appropriate choice, based on the particular OID or OIDs to be written to the tag, possibly combined with other ALE settings that could for example convey the application's desire to use an external Directory structure (or other special features that a DSFID can indicate) with the tag.

If the optional `.force` is not present in the `dataSpec` value, then the ALE implementation SHALL omit all initialization steps as described above if the prior contents of the bits 00h through 07h are non-zero.

Explanation (non-normative); In other words, if `.force` is omitted, then the tag is initialized only if it was not previously initialized to a valid ISO 15962 memory state (though not necessarily having the specified DSFID). Previously existing data sets are preserved in that case. If the tag was not previously initialized, or if `.force` is specified, then the memory is always initialized to a state containing no data sets.

When interacting with a Gen1 Tag, the implementation SHALL raise an "operation not possible" condition. When interacting with any other type of Tag, the interpretation of INITIALIZE on the User memory bank is implementation dependent. An ALE implementation SHOULD carefully document its behavior in this situation.

9.3.6 CCOpDataSpec

The CCOpDataSpec specifies a source of data for a command. A data specification can specify constant data ("literal"), data from an EPC cache (only valid when writing to the EPC bank), data from a named parameter provided as an argument to the `poll` API method, data from an EPC association table, or randomly-generated data.

```
CCOpDataSpec
---
specType : CCOpDataSpecType
data : String
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of a `CCOpDataSpec` as follows.

Field	Type	Description
`specType`	`CCOpDataSpecType`	Specifies what kind of data source provides the data to the command. See Section 9.3.7 and the table below.
`data`	`String`	Further specifies the data source according to the `specType`. See the table below.

Table 65. `CCOpDataSpec` Fields

The ALE implementation SHALL use the following table to determine what data value is used for the command that includes a `CCOpDataSpec`.

Value of `specType`	Data to be used as input to the command
`LITERAL`	The value of the `data` parameter itself, interpreted according to the format implied by the `fieldspec` of the enclosing `CCOpSpec`.
`PARAMETER`	The `value` parameter of the name/value pair occurring in the `CCParameterList` argument to `poll` whose `name` parameter is equal to the `data` parameter of this `CCOpDataSpec`. The `value` parameter of the name/value pair is interpreted according to the format implied by the `fieldspec` of the enclosing `CCOpSpec`.
`CACHE`	The next EPC value taken from the EPC Cache whose name is equal to `data`.

Value of specType	Data to be used as input to the command
ASSOCIATION	The value obtained from looking up the EPC of the Tag being operated upon in the association table whose name is equal to data. The EPC to be used is (a) the EPC that was to be written by the most recent operation in the CCCmdSpec that writes the EPC field; or (b) the EPC read from the Tag, if no operation in this CCCmdSpec prior to this one writes the EPC field.
RANDOM	The next random value generated from the RNG whose name is equal to data.

Table 66. CCOpDataSpec specType Fields

The define and immediate methods SHALL raise a CCSpecValidationException if any of the following are true for a CCOpDataSpec instance, according to the value of specType. In addition, the define and immediate methods SHALL raise a CCSpecValidationException if a CCOpDataSpec instance is supplied but in Table 64 the opType specifies "[must be omitted]" in the fourth column.

Value of specType	Conditions under which a CCValidationException is raised
LITERAL	• The opType is CHECK or INITIALIZE and the specified value is not legal according to Sections 9.3.5.1 and 9.3.5.2 • The opType is LOCK and the specified value is not legal according to Section 9.3.8 • The opType is something other than CHECK, INITIALIZE, or LOCK, and the specified data value is not valid syntax for the datatype and format implied by the fieldspec of the enclosing CCOpSpec.
PARAMETER	• The opType is CHECK, INITIALIZE, or LOCK.
CACHE	• The opType is CHECK, INITIALIZE, or LOCK. • There is no EPC Cache whose name is equal to data. • The datatype implied by the fieldspec of the enclosing CCOpSpec is not epc.

Value of `specType`	**Conditions under which a** `CCValidationException` **is raised**
`ASSOCIATION`	• The `opType` is `CHECK`, `INITIALIZE`, or `LOCK`. • There is no association table whose name is equal to `data`. • The `opType` is `WRITE` or `ADD`, and the datatype of the specified association table is not the same as the datatype implied by the fieldspec of the `opSpec`. • The `opType` is PASSWORD or KILL, and the datatype of the specified association table is not `uint`.
`RANDOM`	• The `opType` is `CHECK`, `INITIALIZE`, or `LOCK`. • There is no RNG whose name is equal to `data`. • The datatype implied by the `fieldspec` of the enclosing `CCOpSpec` is not `uint`.

Table 67. `CCOpDataSpec` Validation Rules

9.3.7 CCOpDataSpecType

<<Enumerated Type>> CCOpDataSpecType
LITERAL
PARAMETER
CACHE
ASSOCIATION
RANDOM
<<extension point>>

9.3.8 CCLockOperation

<<Enumerated Type>> CCLockOperation
UNLOCK
PERMAUNLOCK
LOCK
PERMALOCK
<<extension point>>

The ALE implementation SHALL interpret the data parameter of a LOCK command as follows:

CCLockOperation value	Description
`UNLOCK`	The field is unlocked; subsequent privileged operations on this field may be performed without supplying a password.
`PERMAUNLOCK`	The field is permanently unlocked; subsequent privileged operations on this field may be performed without supplying a password, and any attempt to change the lock status of this field results in a `PERMISSION_ERROR`.
`LOCK`	The field is locked; subsequent privileged operations on this field may be performed only if a password is supplied.
`PERMALOCK`	The field is permanently locked; subsequent privileged operations on this field cannot be performed, and any attempt to change the lock status of this field results in a `PERMISSION_ERROR`.

Table 68. `CCLockOperation` Values

The ALE implementation SHALL interpret "subsequent privileged operations" when interacting with a Gen2 Tag as follows:

Fieldname	Subsequent privileged operations
killPwd accessPwd	Read and Write operations.
epcBank tidBank userBank	Write operations.

Table 69. Meaning of "subsequent privileged operations"

9.3.9 CCStatProfileName

Each valid value of CCStatProfileName names a statistics profile that can be included in an CCReports.

<<Enumerated Type>> CCStatProfileName
<<extension point>>

This specification does not define any statistics profiles for the Writing API. Vendors, however, MAY implement their own proprietary profiles.

9.3.10 Validation of CCSpecs

The define and immediate methods of the ALECC API (Section 9.1) SHALL raise a CCSpecValidationException if any of the following are true:

- The specified specName is an empty string or is not accepted by the implementation according to Section 4.5.
- The logicalReaders parameter of CCSpec is null, omitted, is an empty list, or contains any logical reader names that are not known to the implementation.
- The boundarySpec parameter of CCSpec is null or omitted.
- The cmdSpecs parameter of CCSpec is null, omitted, or empty.
- The duration, repeatPeriod, or noNewTagsInterval parameter of CCBoundarySpec is negative.
- Any element of the startTriggerList or stopTriggerList parameter of CCBoundarySpec does not conform to URI syntax as defined by [RFC2396], or is a URI that is not supported by the ALE implementation. Note that an empty string does not conform to URI syntax as defined by [RFC2396].
- The tagsProcessedCount of CCBoundarySpec is negative.

- No stopping condition apart from `afterError` is specified in `CCBoundarySpec`; *i.e.*, `stopTriggerList` is empty, and neither `duration` nor `tagsProcessedCount` nor `noNewTagInterval` nor any vendor extension stopping condition is specified.
- Any `CCCmdSpec` instance has a `name` that is an empty string or that is not accepted by the implementation according to Section 4.5.
- Two `CCCmdSpec` instances have identical values for their `name` fields.
- The `patList` parameter of any `ECFilterListMember` instance is empty, null, or omitted, or any element of `patList` does not conform to the syntax rules for patterns implied by the specified `fieldspec`.
- The `opType` parameter of a `CCOpSpec` is not one of the standard `opType` values specified in Section 9.3.5, or an implementation-specific value known to the ALE implementation.
- The `opType` parameter of a `CCOpSpec` requires a `fieldspec`, and `fieldspec` is null or omitted.
- The `opType` parameter of a `CCOpSpec` does not require a `fieldspec`, and `fieldspec` is specified.
- The `fieldspec` parameter of a `CCOpSpec` is invalid according to Section 8.2.12.
- The `opType` parameter of a `CCOpSpec` requires a `dataSpec`, and `dataSpec` is null or omitted.
- The `opType` parameter of a `CCOpSpec` does not require a `dataSpec`, and `dataSpec` is specified.
- The `dataSpec` parameter of a `CCOpSpec` is invalid according to Section 9.3.6.
- The `dataSpec` parameter of a `CCOpSpec` specifies a value that is invalid for the specified operation, as specified in Section 9.3.6.
- Two or more `CCOpSpec` instances within the same `CCCmdSpec` instance specify the same (non-empty) `opName`.
- Any value of `CCStatProfileName` is not recognized, or is recognized but the specified statistics report is not supported.

9.4 CCReports

The CCReports object is the output from a command cycle.

```
CCReports
---------------------------------------------
specName: String
date: dateTime
ALEID: String
totalMilliseconds: long
initiationCondition : CCInitiationCondition
initiationTrigger : ECTrigger
terminationCondition: CCTerminationCondition
terminationTrigger : ECTrigger
CCSpec: CCSpec
cmdReports: List<CCCmdReport>
<<extension point>>
---
```

The "meat" of a `CCReports` instance is the ordered list of `CCCmdReport` instances, each corresponding to a `CCReportSpec` instance in the command cycle's `CCSpec`, and appearing in the order corresponding to the `CCSpec`. In addition to the reports themselves, `CCReports` contains a number of "header" fields that provide useful information about the command cycle. The implementation SHALL include these fields according to the following definitions:

Field	**Description**
`specName`	The name of the `CCSpec` that controlled this command cycle. In the case of a `CCSpec` that was requested using the `immediate` method (Section 9.1), this name is one chosen by the ALE implementation.
`date`	A representation of the date and time when the command 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 outside the scope of this specification.
`totalMilliseconds`	The total time, in milliseconds, from the start of the command cycle to the end of the command cycle.

Field	Description
`initiationCondition`	Indicates what kind of event caused the command cycle to initiate: the receipt of an explicit start trigger, the expiration of the repeat period, or a transition to the *requested* state when no start triggers were specified in the `CCSpec`. These correspond to the possible ways of specifying the start of a command cycle as defined in Section 9.3.1.
`initiationTrigger`	If `initiationCondition` is `TRIGGER`, the `ECTrigger` instance corresponding to the trigger that initiated the command cycle; omitted otherwise.
`terminationCondition`	Indicates what kind of event caused the command cycle to terminate: the receipt of an explicit stop trigger, the expiration of the command cycle duration, no Tags being processed for the prescribed amount of time, the "tags processed count" being reached, or an error during processing a Tag. These correspond to the possible ways of specifying the end of a command cycle as defined in Section 9.3.1.
`terminationTrigger`	If `terminationCondition` is `TRIGGER`, the `ECTrigger` instance corresponding to the trigger that terminated the command cycle; omitted otherwise.
`CCSpec`	A copy of the `CCSpec` that generated this `CCReports` instance. Only included if the `CCSpec` has `includeSpecInReports` set to true.

Table 70. `CCReports` Fields

9.4.1 CCInitiationCondition

`CCInitiationCondition` is an enumerated type that describes how a command cycle was started.

<<Enumerated Type>> CCInitiationCondition
TRIGGER
REPEAT_PERIOD
REQUESTED
UNDEFINE
<<extension point>>

The ALE implementation SHALL set the `initiationCondition` field of a `CCReports` instance generated at the conclusion of a command cycle according to the condition that caused the command cycle to start, as specified in the following table.

CCInitiationCondition	**Event causing the command cycle to start**
`TRIGGER`	One of the triggers specified in `startTriggerList` of `CCBoundarySpec` was received.
`REPEAT_PERIOD`	The `repeatPeriod` specified in the `CCBoundarySpec` expired, or the command cycle started immediately after the previous command cycle ended because neither a start trigger nor a repeat period was specified.
`REQUESTED`	The `CCSpec` transitioned from the *unrequested* state to the *requested* state and `startTriggerList` in `CCBoundarySpec` was empty.
`UNDEFINE`	Used when an outstanding `poll` call is terminated due to an `undefine` call, while the CCSpec was in the requested state (that is, before any start condition actually occurred). See Section 5.6.1.

Table 71. `CCInitiationCondition` Values

Each row of this table corresponds to one of the possible start conditions specified in Section 9.3.1.

9.4.2 CCTerminationCondition

`CCTerminationCondition` is an enumerated type that describes how a command cycle was ended.

<<Enumerated Type>> CCTerminationCondition
TRIGGER
DURATION
NO_NEW_TAGS
COUNT
ERROR
UNREQUEST
UNDEFINE
<<extension point>>

The ALE implementation SHALL set the `terminationCondition` field of a `CCReports` instance generated at the conclusion of a command cycle according to the condition that caused the command cycle to end, as specified in the following table.

CCTerminationCondition	**Event causing the command cycle to end**
TRIGGER	One of the triggers specified in `stopTriggerList` of `CCBoundarySpec` was received.
DURATION	The `duration` specified in the `CCBoundarySpec` expired.
NO_NEW_TAGS	No new Tags were processed within the `noNewTagsInterval` specified in the `CCBoundarySpec`.
COUNT	The `tagsProcessedCount` limit specified in the `CCBoundarySpec` was reached.
ERROR	The `afterError` parameter in `CCBoundarySpec` was true and an error was encountered in carrying out a `CCOpSpec` on a Tag.
UNREQUEST	The `CCSpec` transitioned to the *unrequested* state. By definition, this value cannot actually appear in a `CCReports` instance sent to any client.
UNDEFINE	The `CCSpec` was removed by an `undefine` call while in the requested or active state. See Section 5.6.1.

Table 72. `CCTerminationCondition` Values

Each row of this table corresponds to one of the possible stop conditions specified in Section 9.3.1.

9.4.3 CCCmdReport

Each `CCCmdSpec` in the `CCSpec` is associated with a `CCCmdReport`.

CCCmdReport
cmdSpecName: String
tagReports: List<CCTagReport>
<<extension point>>

An ALE implementation SHALL construct a `CCCmdReport` as follows:

Field	Type	Description
cmdSpecName	String	A copy of the `cmdSpecName` field from the corresponding `CCCmdSpec` within the `CCSpec` that controlled this command cycle.
tagReports	List<CCTagReport>	An unordered list of `CCTagReport` instances, one for each Tag processed during the command cycle that matches the filter conditions of the corresponding `CCCmdSpec`.

Table 73. `CCCmdReport` Fields

9.4.4 CCTagReport

A `CCTagReport` describes what happened during the processing of a single Tag.

CCTagReport
id : String
opReports : List<CCOpReport>
stats : List<CCTagStat>
<<extension point>>

An ALE implementation SHALL construct a `CCTagReport` as follows:

Field	Type	Description
id	String	(Optional) A data value that identifies the Tag that was operated upon. When a Tag Protocol normally reports a tag identifier (that is, a data value that serves to distinguish one Tag from another) when operating upon a Tag, the ALE implementation SHOULD include this value here. In particular, when operating upon a Gen2 Tag, an ALE implementation SHOULD include the EPC value read from the Tag during singulation (i.e., before any operations are performed upon the Tag in this command cycle). When the id field is an EPC, it SHALL be reported in epc-tag format.
opReports	List<CCOpReport>	An ordered list of CCOpReport instances, one for each of the corresponding CCOpSpec instances in the corresponding CCCmdSpec, in the corresponding order.
stats	List<CCTagStat>	Null, if the statProfileNames parameter of the corresponding CCCmdSpec is empty, omitted, or null. Otherwise, contains a CCTagStat for each statistics profile named in the statProfileNames parameter of the corresponding CCCmdSpec, in the corresponding order.

Table 74. CCTagReport Fields

9.4.5 CCOpReport

A CCOpReport contains the result of a single CCOpSpec executing on a single Tag during a command cycle.

```
CCOpReport
data : String // Conditional
opStatus : CCStatus
opName : String // Conditional
<<extension point>>
---
```

An ALE implementation SHALL construct a CCOpReport as follows:

Field	Type	Description
data	String	(Conditional) The result of the operation, according to the table below, or null if an error occurred.
opStatus	CCStatus	Specifies whether the operation succeeded or failed (see Section 9.4.6).
opName	String	(Conditional) A copy of the opName parameter of the corresponding CCOpSpec. Omitted if the opName parameter was omitted from the corresponding CCOpSpec.

3478 Table 75. CCOpReport Fields

3479 The value of the data field SHALL be constructed according to the following table:

CCOpType Value	Description	data Value
READ	Read from memory	The value that was read, formatted according to the fieldspec parameter of the corresponding CCOpSpec.
WRITE	Write to memory	The value that was written, formatted according to the fieldspec parameter of the corresponding CCOpSpec.
PASSWORD	Provide a password to enable subsequent commands; for Gen2 Tags, this transitions the tag to the "secured" state.	Null
KILL	Kill a tag; for Gen2 Tags this means to use the Gen2 "kill" command.	Null
LOCK	Sets access permissions for a memory field	Null

3480 Table 76. CCOpReport data Field Values

3481 9.4.6 CCStatus

3482 CCStatus is an enumerated value that lists the several possible outcomes for a given
3483 operation.

<<Enumerated Type>> CCStatus
SUCCESS
MISC_ERROR_TOTAL
MISC_ERROR_PARTIAL
PERMISSION_ERROR
PASSWORD_ERROR
FIELD_NOT_FOUND_ERROR
OP_NOT_POSSIBLE_ERROR
OUT_OF_RANGE_ERROR
FIELD_EXISTS_ERROR
MEMORY_OVERFLOW_ERROR
MEMORY_CHECK_ERROR
ASSOCIATION_TABLE_VALUE_INVALID
ASSOCIATION_TABLE_VALUE_MISSING
EPC_CACHE_DEPLETED
<<extension point>>

The codes that contain `ERROR` in their names are errors that arise during the interaction between the ALE implementation and the Tag. The other codes (apart from `SUCCESS`) result from conditions that can be detected without interacting with the Tag.

An ALE implementation SHALL return `CCStatus` codes according to the following table:

Status Code	Description
SUCCESS	The operation completed successfully.
MISC_ERROR_TOTAL	An error occurred during the processing of this operation that resulted in total failure. The state of the Tag following the operation attempt is unchanged. An ALE implementation SHALL return this code only if no more specific code applies.

Status Code	Description
MISC_ERROR_PARTIAL	An error occurred during the processing of this operation that resulted in partial failure. The state of the Tag following the operation attempt is indeterminate. For example, if a WRITE operation requires issuing two write commands via an RFID Tag's Air Interface, a failure during the second Air Interface command results in partial failure of the overall WRITE operation. An ALE implementation SHALL return this code only if no more specific code applies.
PERMISSION_ERROR	The operation failed because the Tag denied permission: for example, an attempt to write to a locked field of a Gen2 RFID Tag without first supplying an access password. An ALE implementation SHALL return this code only if the denial of permission resulted in total failure.
PASSWORD_ERROR	(PASSWORD operation only) The supplied password was incorrect.
FIELD_NOT_FOUND_ ERROR	The specified field of the Tag was not found (see Section 5.4).
OP_NOT_POSSIBLE_ ERROR	The specified operation is not possible on the specified field of the Tag (see the "operation not possible" condition specified in Section 5.4). In contrast to PERMISSION_ERROR, which indicates an error that could be overcome by supplying appropriate credentials or by an appropriately privileged client, OP_NOT_POSSIBLE_ERROR indicates that limitations of the Tag or the ALE implementation prevent this operation from being carried out on the specified field under any circumstances.
OUT_OF_RANGE_ERROR	The specified value could not be encoded using the available number of bits (see the "out of range" condition specified in Section 5.4). This applies to the WRITE and ADD operations for fixed fields, as well as to the PASSWORD and KILL operations.
FIELD_EXISTS_ERROR	The ADD operation failed because the specified field already exists in memory. This error cannot occur for a fixed field fieldspec.

Status Code	Description
MEMORY_OVERFLOW_ ERROR	Attempting to add a new field or modify an existing variable-length field to the memory bank would overflow the free memory left in the memory bank.
MEMORY_CHECK_ERROR	The CHECK operation failed.
ASSOCIATION_TABLE_ VALUE_INVALID	The value retrieved from the association table was not valid syntax for the datatype and format implied by the `fieldspec` parameter of the `CCOpSpec`.
ASSOCIATION_TABLE_ VALUE_MISSING	The association table did not contain a value for the EPC read from the Tag.
EPC_CACHE_DEPLETED	The specified EPC Cache was empty at the time of the operation attempt.

Table 77. `CCStatus` Values

Explanation (non-normative): The ALE specification only provides for a status code to be returned on a per operation basis. If an implementation wants to report additional information about a particular operation, it may do so through vendor extension or through out-of-band mechanisms such as logging.

9.4.7 CCTagStat

A `CCTagStat` provides additional, implementation-defined information about each "sighting" of a Tag, that is, each time a Tag is acquired by one of the Readers participating in the command cycle.

```
CCTagStat
profile : CCStatProfileName
statBlocks : List<ECReaderStat>
---
```

An ALE implementation SHALL construct a `CCTagStat` as follows:

Field	Type	Description
`profile`	`CCStatProfileName`	The name of the statistics profile that governed the generation of this `CCTagStat` instance.
`statBlocks`	`List<ECReaderStat>`	An unordered list containing an `ECReaderStat` instance for each Reader that sighted this Tag.

Table 78. `CCTagStat` Fields

Note that `CCTagStat` is identical to `ECTagStat` (Section Table 52), except that the `profile` parameter is an instance of `CCStatProfileName` instead of

ECStatProfileName. The remaining content shares the ECReaderStat and ECSightingStat classes defined in the Reading API.

9.5 EPCCache

An EPCCache is a set of EPC values maintained by the ALE implementation, used to provide a value to the WRITE command in CCOpSpec (see Section 9.3.4). ALE clients define and maintain EPCCaches through the following API methods, which are part of the ALECC interface.

```
<<interface>>
ALECC
[Continued from Section 9.1]
---
defineEPCCache(cacheName : String, spec : EPCCacheSpec,
replenishment : EPCPatternList) : void
undefineEPCCache(cacheName : String) : EPCPatternList
getEPCCache(cacheName : String) : EPCCacheSpec
getEPCCacheNames() : List<String> // returns a list of
cacheNames as strings
replenishEPCCache(cacheName : String, replenishment :
EPCPatternList) : void
depleteEPCCache(cacheName : String) : EPCPatternList
getEPCCacheContents(cacheName : String) : EPCPatternList
<<extension point>>
```

An ALE implementation SHALL implement the above methods of the ALE Writing API as specified in the following table:

Method	Description
defineEPCCache	Creates an EPC Cache whose name is cacheName, with initial contents as specified by replenishment. The spec parameter, if non-null, specifies implementation-specific parameters that control the operation of the EPC Cache. If spec is null, the implementation SHALL use default settings for any controls of this kind.
undefineEPCCache	Removes the EPC Cache whose name is cacheName. The remaining contents of the EPCCache at the time of removal is returned.
getEPCCache	Returns the (possibly null) value of the spec parameter that was provided to the defineEPCCache method at the time the EPC Cache was created.

Method	Description
`getEPCCacheNames`	Returns an unordered list of the names of all currently defined EPC Caches.
`replenishEPCCache`	Appends `replenishment` to the end of the current contents of the EPC Cache named `cacheName`.
`depleteEPCCache`	Removes all EPCs from the EPC Cache named `cacheName`, and returns an `EPCPatternList` instance to the caller that enumerates the EPCs that were in the cache at the time they were removed.
`getEPCCacheContents`	Returns an `EPCPatternList` instance that enumerates the EPCs currently in the EPC Cache named `cacheName`.

Table 79. ALECC Interface Methods (continued from Table 56)

The implementation SHALL maintain each defined EPC Cache in the following manner. An EPC Cache is an ordered list of EPCs, whose initial contents is specified by the `replenishment` argument to `defineEPCCache`. The EPC Cache may be referred to by name in a `CCOpDataSpec` whose `specType` is equal to `CACHE`. Each time during a command cycle that a Tag is processed using that `CCOpDataSpec`, the first element of the EPC Cache is removed and used as the value for the operation specified in the `CCOpSpec`. If there is no first element (because the EPC Cache is empty), then the operation results in an `EPC_CACHE_DEPLETED` error that is reported in the `CCOpReport` for that Tag. At any time, the ALE client may add more EPCs to the end of list by invoking the `replenishEPCCache` method.

The ALE implementation may represent the list of EPCs in any manner it wishes, so long as the net result is equivalent to the description above. In particular, it may maintain the state of the list in any suitable store, including an external store.

9.5.1 Exceptions

Methods of the ALE Writing API defined in Section 9.5 signal error conditions to the client by means of exceptions, some of which are specified in Section 9.1.1, others as specified below.

Exception Name	Meaning
`EPCCacheSpec-ValidationException`	The specified `EPCCacheSpec` is not valid. The specific conditions under which this exception is raised are vendor specific. This exception SHALL NOT be raised, however, if the `spec` argument to `defineEPCCache` is null, or if the implementation has not made any extensions to `EPCCacheSpec`. Moreover, all implementations SHALL raise this exception if the specified `cacheName` is an empty string or is not accepted by the implementation according to Section 4.5.
`InvalidPatternException`	The `replenishment` parameter of `defineEPCCache` or `replenishEPCCache` is invalid.
`InUseException`	The specified EPC Cache cannot be undefined, because there exist one or more CCSpecs that refer to it.

Table 80. Exceptions in the ALECC Interface (continued from Table 57)

The exceptions that may be raised by each Writing API method from Section 9.5 are indicated in the table below. An ALE implementation SHALL raise the appropriate exception listed below when the corresponding condition described above and in Section 9.1.1 occurs. If more than one exception condition applies to a given method call, the ALE implementation may raise any of the exceptions that applies.

ALE Method	Exceptions

`defineEPCCache`	`DuplicateNameException`
	`EPCCacheSpecValidationException`
	`InvalidPatternException`
	`SecurityException`
	`ImplementationException`

`undefineEPCCache`	`NoSuchNameException`
	`InUseException`
	`SecurityException`
	`ImplementationException`

`getEPCCache`	`NoSuchNameException`
	`SecurityException`
	`ImplementationException`

`getEPCCacheNames`	`SecurityException` `ImplementationException`

ALE Method	Exceptions
replenishEPCCache	NoSuchNameException InvalidPatternException SecurityException ImplementationException
depleteEPCCache	NoSuchNameException SecurityException ImplementationException
getEPCCacheContents	NoSuchNameException SecurityException ImplementationException

Table 81. Exceptions Raised by each ALECC Interface Method (continued from Table 58)

9.5.2 EPCCacheSpec

The `EPCCacheSpec` class contains only an extension point. Implementations MAY define extensions to this class in order to provide additional parameters to control the behavior of an EPC Cache. For example, if an implementation wishes to provide a mechanism to notify clients when an EPC Cache is nearing empty, it may use extensions to `EPCCacheSpec` to configure this mechanism, such as providing an address for sending notifications, a threshold level for notification, and so on.

EPCCacheSpec
<<extension point>>

9.5.3 EPCPatternList

An `EPCPatternList` specifies an ordered list of EPCs, using EPC pattern syntax.

EPCPatternList
patterns : List<String>

An ALE implementation SHALL interpret the fields of `EPCPatternList` as follows:

Field	Type	Description
patterns	List<String>	An ordered list, each of which is an EPC pattern URI as defined in [TDS1.3.1] containing at most one field that is a `[lo-hi]` range or a `*` wildcard, which field must be numeric. The interpretation of these patterns is specified below.

Table 82. `EPCPatternList` Fields

An ALE implementation SHALL interpret each EPC pattern URI element of `patterns` as denoting an ordered list of individual EPCs obtained by enumerating in ascending numerical order all EPCs that match the pattern. An ALE implementation SHALL interpret the overall `EPCPatternList` instance as denoting an ordered list of individual EPCs obtained by concatenating, in order, the EPCs denoted by each EPC pattern URI element.

Example (non-normative): For example, an `EPCPatternList` *containing the following three pattern URIs:*

```
urn:epc:pat:sgtin-96:0.0614141.112345.[0-2]
urn:epc:pat:sgtin-96:0.0614141.112345.100
urn:epc:pat:sgtin-96:0.0614141.112345.[1000-1001]
```

denotes the following list of six EPCs:

```
urn:epc:tag:sgtin-96:0.0614141.112345.0
urn:epc:tag:sgtin-96:0.0614141.112345.1
urn:epc:tag:sgtin-96:0.0614141.112345.2
urn:epc:tag:sgtin-96:0.0614141.112345.100
urn:epc:tag:sgtin-96:0.0614141.112345.1000
urn:epc:tag:sgtin-96:0.0614141.112345.1001
```

Note that wildcard fields must be numeric, so that the following pattern URI is not valid:

```
urn:epc:tag:sgtin-198:0.0614141.112345.*
```

Because the serial number (rightmost) field of an SGTIN-198 EPC is alphanumeric, it may not be used as a pattern for an `EPCPatternList`*. If specified it will lead to an* `InvalidPatternException`*.*

9.6 AssociationTable

An association table provides a list of name-value pairs where the name is an EPC and the value is a string. These tables are maintained by the ALE implementation and used to provide the appropriate value to the `WRITE`, `PASSWORD and KILL` commands in a `CCOpSpec` (see Section 9.3.4). ALE clients define and maintain association tables through the following methods, which are part of the `ALECC` interface.

```
<<interface>>
ALECC
[continued from Section 9.1]
---
defineAssocTable(tableName : String, spec : AssocTableSpec,
entries : AssocTableEntryList) : void
undefineAssocTable(tableName : String) : void
getAssocTableNames() : List<String> // returns a list of
tableNames as strings
getAssocTable(tableName : String) : AssocTableSpec
putAssocTableEntries(tableName : String, entries :
AssocTableEntryList) : void
getAssocTableValue(tableName : String, epc : String) :
String
getAssocTableEntries(tableName : String, patList :
EPCPatternList) : AssocTableEntryList
removeAssocTableEntry(tableName : String, epc : String) :
void
removeAssocTableEntries(tableName : String, patList :
EPCPatternList) : void
<<extension point>>
```

An ALE implementation SHALL implement the above methods of the ALE Writing API as specified in the following table:

Method	Description
`defineAssocTable`	Creates an EPC Association Table whose name is `tableName`, with initial contents as specified by `entries`. The `spec` parameter specifies the datatype and format for values in the association table.
`undefineAssocTable`	Deletes the EPC Association Table whose name is `tableName`.
`getAssocTableNames`	Returns an unordered list of the names of all defined EPC Association Tables.
`getAssocTable`	Returns the `AssocTableSpec` that was specified when the table whose name is `tableName` was defined.

Method	Description
`putAssocTableEntries`	Adds or replaces entries in the EPC Association Table whose name is `tableName`, according to `entries`. For each member of `entries` that specifies a key that is not currently in the table, a new entry is created. For each member of `entries` that specifies a key that is currently in the table, the value for the entry is replaced.
`getAssocTableValue`	Returns the value currently associated with the specified `epc` in the EPC Association Table named `tableName`, in the format specified when the table was defined, or null if no entry is defined for that EPC.
`getAssocTableEntries`	Returns an `AssocTableEntryList` containing an entry for each EPC that matches one of the patterns specified in `patList` and has an entry in the EPC Association Table named `tableName`. If no entries match the specified patterns, an `AssocTableEntryList` containing zero entries is returned. The value field of each entry returned SHALL be in the format specified when the table was defined.
`removeAssocTableEntry`	Removes the entry for `epc` in the EPC Association Table named `tableName`, if such an entry exists. Otherwise, does nothing.
`removeAssocTableEntries`	Removes the entries for any EPC in the EPC Association Table named `tableName` that matches one of the patterns specified in `patList`. If no entries match the patterns, does nothing.

Table 83. ALECC Interface Methods (continued from Table 79)

The ALE implementation may represent an association table in any manner it wishes, so long as the net result is equivalent to the description above. In particular, it may maintain the state of a table in any suitable store, including an external store.

9.6.1 Exceptions

Methods of the ALE Writing API defined in Section 9.6 signal error conditions to the client by means of exceptions, some of which are specified in Section 9.1.1, others as specified below.

Exception Name	Meaning
`AssocTableValidationException`	The `spec` parameter of `defineAssocTable` is invalid, as specified in Section 9.6.2 or the `tableName` parameter is an empty string or is not accepted by the implementation according to Section 4.5.
`InvalidPatternException` (same exception as defined in Section 9.5.1)	The `patList` parameter of `getAssocTableEntries` or `removeAssocTableEntries` is invalid.
`InvalidEPCException`	The specified `epc` parameter is not valid syntax for one of the EPC data type formats indicated as "RW" in the table in Section 6.2.1.1.
`InvalidAssocTableEntry-Exception`	The `entries` parameter of `defineAssocTable` or `putAssocTableEntries` contains two entries having the same key, contains a key that is not valid syntax for one of the EPC data type formats indicated as "RW" in the table in Section 6.2.1.1, or contains a value that is not valid syntax for the datatype and format specified when the table was defined. In the event this exception is raised, the ALE implementation SHALL NOT define a new association table nor modify an existing association table, even if some entries in the `entries` parameter were valid.
`InUseException` (Same exception as defined in Section 9.5.1.)	The specified Association Table cannot be undefined, because there exist one or more CCSpecs that refer to it.

Table 84. Exceptions in the ALECC Interface (continued from Table 80)

The exceptions that may be raised by each Writing API method from Section 9.6 are indicated in the table below. An ALE implementation SHALL raise the appropriate exception listed below when the corresponding condition described above and in Section 9.1.1 occurs. If more than one exception condition applies to a given method call, the ALE implementation may raise any of the exceptions that applies.

ALE Method	Exceptions

`defineAssocTable`	`DuplicateNameException`
	`AssocTableValidationException`
	`InvalidAssocTableEntryException`
	`SecurityException`
	`ImplementationException`

`undefineAssocTable`	`NoSuchNameException`
	`InUseException`
	`SecurityException`
	`ImplementationException`

`getAssocTableNames`	`SecurityException` `ImplementationException`

`getAssocTable`	`NoSuchNameException`
	`SecurityException`
	`ImplementationException`

`putAssocTableEntries`	`NoSuchNameException`
	`InvalidAssocTableEntryException`
	`SecurityException`
	`ImplementationException`

`getAssocTableValue`	`NoSuchNameException`
	`InvalidEPCException`
	`SecurityException`
	`ImplementationException`

`getAssocTableEntries`	`NoSuchNameException`
	`InvalidPatternException`
	`SecurityException`
	`ImplementationException`

`removeAssocTableEntry`	`NoSuchNameException`
	`SecurityException`
	`InvalidEPCException`
	`ImplementationException`

`removeAssocTableEntries`	`NoSuchNameException`
	`InvalidPatternException`
	`SecurityException`
	`ImplementationException`

Table 85. Exceptions Raised by each ALECC Interface Method (continued from Table 81)

9.6.2 AssocTableSpec

The `AssocTableSpec` class specifies the datatype and format for entries in an association table. Implementations MAY define extensions to this class in order to provide additional parameters to control the behavior of an Association Table, such as connections to external storage, etc.

```
AssocTableSpec
---
datatype : String
format : String
<<extension point>>
---
```

An ALE implementation SHALL interpret an `AssocTableSpec` instance as follows:

Field	Type	Description
`datatype`	`String`	Specifies what kind of data values the association table holds.
`format`	`String`	Specifies the syntax used to present table values through the methods specified in Section 9.6.

Table 86. `AssocTableSpec` Fields

The `defineAssocTable` method SHALL raise an `AssocTableValidationException` if any of the following are true:

- The value of `datatype` is not a valid datatype as specified in Section 6.2 or a datatype recognized as a vendor extension.
- The value of `format` is not a valid format for the specified `datatype`.

9.6.3 AssocTableEntryList

An `AssocTableEntryList` provides the list of specific key/value pairs utilized by an EPC Association Table.

```
AssocTableEntryList
---
entries : List<AssocTableEntry>
---
```

An ALE implementation SHALL interpret the fields of `AssocTableEntryList` as follows:

Field	Type	Description
`entries`	`List<AssocTableEntry>`	An unordered list of `AssocTableEntry` instances.

Table 87. `AssocTableEntryList` Fields

9.6.4 AssocTableEntry

An `AssocTableEntry` is a single key/value pair within an EPC Association Table.

```
AssocTableEntry
key : String
value : String
---
```

An ALE implementation SHALL interpret the fields of `AssocTableEntry` as follows:

Field	Type	Description
`key`	`String`	The EPC for which this entry supplies the associated value.
`value`	`String`	The value associated with the key, in the syntax specified when the table was defined.

Table 88. `AssocTableEntry` Fields

9.7 Random Number Generator

A Random Number Generator (RNG) provides a source of random numbers that can be used by the `WRITE` command in a `CCOpSpec` (see Section 9.3.4). ALE clients define and maintain random number generators through the following methods, which are part of the `ALECC` interface.

```
<<interface>>
ALECC
[continued from Section 9.1]
---
defineRNG(rngName : String, rngSpec : RNGSpec) : void
undefineRNG(rngName : String) : void
getRNGNames() : List<String> // returns a list of rngNames
as strings
getRNG(rngName : String) : RNGSpec
<<extension point>>
```

An ALE implementation SHALL implement the above methods of the ALE Writing API as specified in the following table:

Method	Description
`defineRNG`	Creates a random number generator whose name is `rngName`. The `rngSpec` parameter specifies the range of the random numbers to be generated.

Method	Description
undefineRNG	Deletes the random number generator whose name is rngName.
getRNGNames	Returns an unordered list of the names of all defined random number generators.
getRNG	Returns the value of the rngSpec parameter that was provided to the defineRNG method at the time the random number generator was created.

Table 89. ALECC Interface Methods (continued from Table 83)

9.7.1 Exceptions

Methods of the ALE Writing API defined in Section 9.7 signal error conditions to the client by means of exceptions, some of which are specified in Section 9.1.1, others as specified below.

Exception Name	Meaning
RNGValidationException	The specified RNGSpec is not valid according to Section 9.7.2. Moreover, all implementations SHALL raise this exception if the specified rngName is an empty string or is not accepted by the implementation according to Section 4.5.
InUseException (Same exception as defined in Section 9.5.1.)	The specified random number generator cannot be undefined, because there exist one or more CCSpecs that refer to it.

Table 90. Exceptions in the ALECC Interface (continued from Table 84)

The exceptions that may be raised by each Writing API method from Section 9.7 are indicated in the table below. An ALE implementation SHALL raise the appropriate exception listed below when the corresponding condition described above and in Section 9.1.1 occurs. If more than one exception condition applies to a given method call, the ALE implementation may raise any of the exceptions that applies.

ALE Method	Exceptions
defineRNG	DuplicateNameException RNGValidationException SecurityException ImplementationException
undefineRNG	NoSuchNameException InUseException SecurityException ImplementationException

ALE Method	Exceptions
getRNGNames	SecurityException ImplementationException
getRNG	NoSuchNameException SecurityException ImplementationException

Table 91. Exceptions Raised by each ALECC Interface Method (continued from Table 85)

9.7.2 RNGSpec

The RNGSpec class specifies the range of random numbers that should be generated by the random number generator.. Implementations MAY define extensions to this class in order to provide additional parameters to control the behavior of a random number generator. This may include, for example, parameters to set an initial seed, parameters to govern the use of a hardware random number genereator, etc. Implementations SHALL provide documentation specifying both how the parameters are interpreted by defineRNG and how the parameters are set when returned from getRNG.

RNGSpec
length : Integer
<<extension point>>

An ALE implementation SHALL interpret an RNGSpec instance as follows:

Field	Type	Description
length	Integer	The number of bits for the random numbers generated by this random number generator. Random numbers SHALL be in the range 0 through $2^{length}-1$, inclusive.

Table 92. RNGSpec Fields

The defineRNG method SHALL raise an RNGValidationException if length is not a positive integer.

9.8 ALECCCallback Interface

The ALECCCallback interface is the path by which an ALE implementation delivers asynchronous results from command cycles to subscribers.

```
<<interface>>
ALECCCallback
---
callbackResults(reports : CCReports) : void
```

Referring to the state transition tables in Section 5.6.1, whenever a transition specifies that “reports are delivered to subscribers” the ALE implementation SHALL attempt to deliver the results to each subscriber by invoking the `callbackResults` method of the `ALECCCallback` interface once for each subscriber, passing the `CCReports` for the command cycle as specified above, and using the binding and addressing information specified by the notification URI for that subscriber as specified in the `subscribe` call. All subscribers receive an identical `CCReports` instance.

Explanation (non-normative): The ***`ALECCCallback`*** *interface is defined very simply, to allow for a wide variety of possible implementations. A binding of the* ***`ALECCCallback`*** *interface may not be a request-response style RPC mechanism at all, but may instead just be a one-way message transport, where the message payload is the* ***`CCReports`*** *instance. Indeed, this is true of all of the standardized bindings of this interface described in Part II [ALE1.1Part2].*

10 ALE Logical Reader API

The ALE Logical Reader API is an interface through which clients may define logical reader names for use with the Reading API and the Writing API, each of which maps to one or more sources/actuators provided by the implementation. The API also provides for the manipulation of configuration properties associated with logical reader names. The available properties and their meanings are implementation-specific; however, this specification defines a small set of standardized properties that may be used to configure “smoothing” behavior for reading Tags. The specification of the Logical Reader API follows the general rules given in Section 4.

10.1 Background (non-normative)

In specifying an event cycle or command cycle, an ALE client names one or more channels through which Tags are accessed. This is usually necessary, as an ALE implementation may manage many devices 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 devices are to be used for a given event cycle or command 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 devices deployed in the field are likely to be unknown at the time an application is authored and configured.
- If a device is replaced, this unique reader device identity will change, forcing the application configuration to be changed.
- If the number of devices must change – *e.g.*, because it is discovered that four RFID 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 devices assigned to fulfill that purpose. Any ALE ECSpec or CCSpec that refers to `DockDoor42` is understood by the ALE implementation to refer to the physical device (or devices) associated with that name.

Logical reader 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 RFID 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 reader names to physical reader devices, synthetic readers, and other sources of EPC events. Configuration information of this kind may be established through static configuration, vendor-specific APIs, dynamic discovery mechanisms, or other methods. These are key extensibility points. While implementations are likely to vary widely in the methods and types of physical device configuration they provide, a very common requirement is to introduce a logical reader name as simple alias for one or more other logical reader names. For example, an implementation may provide an implementation-specific way to configure logical reader names for individual antennas of physical reader devices, but then a user may wish to define a logical reader name like `DockDoor42` as an alias for three particular logical reader names associated with individual antennas. The Logical Reader API is intended to provide a standardized way to meet that requirement.

10.2 ALE Logical Reader API

The Logical Reader API specified in the following subsections provides a standardized way for an ALE client to define a new logical reader name as an alias for one or more other logical reader names. The API also provides for manipulating "properties" (name/value pairs) associated with a logical reader name. Finally, the API provides a means for a client to get a list of all of the logical reader names that are available, and to learn certain information about each logical reader.

Defining a new logical reader name as an alias for one or more other logical reader names is not useful unless there exist some logical reader names to begin with. Ultimately, there must be some logical reader names that correspond to actual channels for manipulating

Tags, that are not themselves aliases for other logical readers. Within this Section 10, the term "composite reader" refers to a logical reader name that has been defined as an alias for other logical reader names, and the term "base reader" refers to a logical reader name that is not defined as an alias, and instead corresponds to an actual channel for manipulating Tags.

Implementations may vary widely as to how base readers come into existence. For example, an ALE implementation that is embedded in a four-antenna RFID reader may provide four fixed logical reader names, one for each antenna. These names exist without the ALE client making any calls to the Logical Reader API. Another example is a software implementation of ALE that is designed to interface to many different RFID readers and other devices via a network; such software may provide a means for users to configure a new base logical reader name by specifying the device make and model, network address, and other configuration parameters. Because this specification does not provide a standardized way to configure base readers, some implementations may provide a means outside of the ALE API for configuring base readers, while others may use vendor extensions to the Logical Reader API for this purpose.

In summary, there are three ways that logical readers may come into existence:

- *Composite Reader* A composite reader is a logical reader that is defined by an ALE client, using the Logical Reader API, as an alias for other logical reader names, which themselves may be composite readers or base readers.

- *Externally-defined Base Reader* An externally-defined base reader is a logical reader that is an actual channel for manipulating Tags, and that is defined by means outside the Logical Reader API. How such logical readers are defined is implementation-specific. Subsequently, an implementation may, through vendor extensions, allow a client to retrieve or change the configuration of an externally-defined base reader. See Section 10.3.2.

- *API-defined Base Reader* An API-defined base reader is a logical reader that is an actual channel for manipulating Tags, and that is defined through the Logical Reader API. Because the Logical Reader API does not provide a standardized way of defining base readers, an API-defined base reader can only be created through the use of vendor extensions. See Section 10.3.2.

The conformance requirements in Section 10.3.2 specify which of these possibilities an implementation must support.

10.3 API

```
<<interface>>
ALELR
---
define(name : String, spec : LRSpec) : void
update(name : String, spec : LRSpec) : void
undefine(name : String) : void
getLogicalReaderNames() : List<String>
getLRSpec(name : String) : LRSpec
addReaders(name : String, readers : List<String>) : void
setReaders(name : String, readers : List<String>) : void
removeReaders(name : String, readers : List<String>) : void
setProperties(name : String, properties : List<LRProperty>) : void
getPropertyValue(name : String, propertyName : String) : String
getStandardVersion() : String
getVendorVersion() : String
<<extension point>>
```

An ALE implementation SHALL implement the methods of the ALE Logical Reader API as specified in the following table:

Method	Description
`define`	Creates a new logical reader named `name` according to `spec`.
`update`	Changes the definition of the logical reader named `name` to match the specification in the `spec` parameter. This is different than calling `undefine` followed by `define`, because `update` may be called even if there are defined ECSpecs, CCSpecs, or other logical readers that refer to this logical reader.
`undefine`	Removes the logical reader named `name`.
`getLogicalReaderNames`	Returns an unordered list of the names of all logical readers that are visible to the caller. This list SHALL include both composite readers and base readers.

Method	Description
`getLRSpec`	Returns an `LRSpec` that describes the logical reader named `name`. See Section 10.3.2 for conformance requirements regarding what information is included in the `LRSpec`.
`addReaders`	Adds the specified logical readers to the list of component readers for the composite logical reader named `name`. This is equivalent to calling `getLRSpec`, modifying the `LRSpec` that is returned to include the specified logical readers in the reader list, and then calling `update` with the modified `LRSpec`.
`setReaders`	Changes the list of component readers for the composite logical reader named `name` to the specified list. This is equivalent to calling `getLRSpec`, modifying the `LRSpec` that is returned by replacing the reader list with the specified list of logical readers, and then calling `update` with the modified `LRSpec`.
`removeReaders`	Removes the specified logical readers from the list of component readers for the composite logical reader named `name`. Any reader name within `readers` that is not currently among the component readers of the specified logical reader is ignored. This is equivalent to calling `getLRSpec`, modifying the `LRSpec` that is returned by removing any references to logical readers in the specified reader list, and then calling `update` with the modified `LRSpec`.
`setProperties`	Changes properties for the logical reader named `name` to the specified list. This is equivalent to calling `getLRSpec`, modifying the properties in the `LRSpec` according to the table below, and then calling `update` with the modified `LRSpec`.
`getPropertyValue`	Returns the current value of the specified property for the specified reader, or null if the specified reader does not have a property with the specified name.
`getStandardVersion`	Returns a string that identifies what version of the specification this implementation of the ALE Logical Reader API complies with, as specified in Section 4.3.
`getVendorVersion`	Returns a string that identifies what vendor extensions of the ALE Logical Reader API this implementation provides, as specified in Section 4.3.

Table 93. ALELR Interface Methods

The `setProperties` method SHALL modify the properties of a logical reader according to the following table. For each property, the table specifies the state of that property following the call to `setProperties`, as a function of its former state and the `properties` parameter to `setProperties`.

	Logical Reader formerly did not have property X	Logical Reader formerly did have property X
`properties` parameter to `setProperties` does not include property X	No change: logical reader does not have property X	No change: logical reader continues to have property X with the same value.
`properties` parameter to `setProperties` includes property X, with a null value	No change: logical reader does not have property X	Logical reader no longer has property X.
`properties` parameter to `setProperties` includes Property X, with a non-null value	Logical reader now has property X, with value as specified in `properties` parameter to `setProperties`.	Logical reader continues to have property X, with value changed to be as specified in `properties` parameter to `setProperties`.

Table 94. Behavior of the `setProperties` Method of the ALELR Interface

The `update`, `addReaders`, `setReaders`, `removeReaders`, and `setProperties` methods are intended to allow the definition of a logical reader to be changed without requiring the client to undefine the reader and then define it again. This allows these methods to be called even if there exist ECSpecs, CCSpecs, or other LRSpecs that refer to the logical reader being changed. Not all implementations, however, may support using these methods to change the definition of a logical reader that is used by an ECSpec or CCSpec that is active at the time the method is called. The five methods named above MAY raise an `InUseException` if at the time the method is called there is an ECSpec or CCSpec in the *active* state that includes the specified logical reader, either directly or indirectly through a composite reader. When an implementation does not raise the `InUseException` in this situation, it is implementation defined as to exactly when the change takes effect, but the change SHOULD take place as soon as possible. An ALE implementation SHALL provide documentation of what "as soon as possible" means; for example, saying that "as soon as possible" means that the change takes place during an event or command cycle, or waits until the conclusion of any active event or command cycles, or whatever is appropriate. These methods SHALL NOT raise an `InUseException`, however, if there are no such active ECSpecs or CCSpecs at the time the method is called – an implementation must be prepared to handle these methods when a logical reader is not actively being used.

The `undefine` method SHALL raise an `InUseException` if there exist one or more ECSpecs, CCSpecs, or other LRSpecs that refer to it, whether ECSpecs or CCSpecs are in the active state or not. This is because the logical reader name does not exist following an `undefine` call, and so if allowed to proceed it would leave the ECSpec, CCSpec, or LRSpec in an inconsistent state.

10.3.1 Error Conditions

Methods of the Logical Reader 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
`DuplicateNameException`	The specified logical reader name already exists.
`NoSuchNameException`	The specified logical reader name does not exist.
`InUseException`	For the `undefine` method, the specified logical reader cannot be undefined, as there exist one or more ECSpecs, CCSpecs, or other LRSpecs that refer to it. For the `update`, `addReaders`, `setReaders`, `removeReaders`, and `setProperties` methods, the specified logical reader cannot be undefined as there exist one or more ECSpecs or CCSpecs in the *active* state that refer to it (directly or indirectly through composite readers), and the implementation does not support changing the logical reader at such times.

Exception Name	Meaning
`ValidationException`	For the `define` method, the specified LRSpec is invalid according to Section 10.4 or the specified `name` is an empty string or is not accepted by the implementation according to Section 4.5. For the `update` method, the specified LRSpec is invalid according to Section 10.4. For the `addReaders` or `setReaders` method, the specified list of readers includes a logical reader name that does not exist. For the `setProperties` method, the specified list of properties includes a property name that is not recognized by the implementation or whose value is not permitted to be changed, or the specified value for a property is not legal for that property name.
`ImmutableReaderException`	The specified externally-defined base reader may not be updated, undefined, or have the specified properties changed. This exception SHALL NOT be raised for a composite reader or an API-defined base reader.
`NonCompositeReaderException`	The specified reader on which the `addReaders` or `setReaders` operation is not a composite reader.
`ReaderLoopException`	The operation, if completed, would have resulted in a composite logical reader directly or indirectly including itself as a component.

Exception Name	Meaning
`SecurityException`	The operation was not permitted due to an access control violation or other security concern. If the Logical Reader API implementation is associated with an implementation of the Access Control API (Section 11), the Logical Reader API implementation SHALL raise this exception if the client was not granted access rights to the called method as specified in Section 11. Other, implementation-specific circumstances may cause this exception; these are outside the scope of this specification.
`ImplementationException`	A generic exception raised 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.

Table 95. Exceptions in the ALELR Interface

The exceptions that may be raised by each Logical Reader API method are indicated in the table below. An ALE implementation SHALL raise the appropriate exception listed below when the corresponding condition described above occurs. If more than one exception condition applies to a given method call, the ALE implementation may raise any of the exceptions that applies.

ALE Method	Exceptions

`define`	`DuplicateNameException`
	`ValidationException`
	`SecurityException`
	`ImplementationException`

ALE Method	Exceptions
update	NoSuchNameException ValidationException InUseException ImmutableReaderException ReaderLoopException SecurityException ImplementationException
undefine	NoSuchNameException InUseException ImmutableReaderException SecurityException ImplementationException
getLogicalReaderNames	SecurityException ImplementationException
getLRSpec	NoSuchNameException SecurityException ImplementationException
addReaders	NoSuchNameException ValidationException InUseException ImmutableReaderException NonCompositeReaderException ReaderLoopException SecurityException ImplementationException
setReaders	NoSuchNameException ValidationException InUseException ImmutableReaderException NonCompositeReaderException ReaderLoopException SecurityException ImplementationException
removeReaders	NoSuchNameException InUseException ImmutableReaderException NonCompositeReaderException SecurityException ImplementationException

ALE Method	Exceptions

`setProperties`	`NoSuchNameException`
	`ValidationException`
	`InUseException`
	`ImmutableReaderException`
	`SecurityException`
	`ImplementationException`

`getPropertyValue`	`NoSuchNameException`
	`SecurityException`
	`ImplementationException`

`getStandardVersion`	`ImplementationException`
`getVendorVersion`	`ImplementationException`

Table 96. Exceptions Raised by each ALELR Interface Method

10.3.2 Conformance Requirements

An implementation of the Logical Reader API SHALL implement all of the methods defined in Section 10.3. In addition, the following conformance requirements that depend on the type of logical reader apply:

Reader Type	Definition	Modification	Introspection
Composite Reader	The implementation SHALL allow a new composite reader to be defined using the `define` method.	The implementation SHALL allow a composite reader to be modified or removed by the `update`, `undefine`, `addReaders`, `setReaders`, `removeReaders`, and `setProperties` methods.	The implementation SHALL include the composite reader's name in the result of `getLogical-ReaderNames`. The implementation SHALL return an `LRSpec` from `getLRSpec` that includes the underlying logical readers and all properties that have been defined.

Reader Type	Definition	Modification	Introspection
Externally-defined Base Reader	(Not applicable – by definition an externally-defined base reader is defined by some means other than the `define` method.)	The implementation MAY allow an externally-defined base reader to be modified or removed by the `update`, `undefine`, and `setProperties` methods. If not, the implementation SHALL raise `ImmutableReader-Exception` for any method it does not permit.	The implementation SHALL include the base reader's name in the result of `getLogical-ReaderNames`. The implementation SHALL return an `LRSpec` from `getLRSpec` that includes any properties that have been defined through the Logical Reader API. The implementation MAY also include in the `LRSpec` any properties or other vendor extensions that provide additional configuration information about the reader.
API-defined Base Reader	The implementation MAY allow a new base reader to be defined using the `define` method. The implementation will likely require vendor-specific properties and/or vendor extensions to `LRSpec` to make this possible.	The implementation SHALL allow an API-defined base reader to be modified or removed by the `update`, `undefine`, and `setProperties` methods, using the same vendor-specific properties and/or vendor extensions to `LRSpec` that were used in the `define` method.	The implementation SHALL include the base reader's name in the result of `getLogical-ReaderNames`. The implementation SHALL return an `LRSpec` from `getLRSpec` that includes all properties and vendor extensions to `LRSpec` that have been defined through the Logical Reader API.

Table 97. Conformance Requirements for ALELR Interface Methods

As indicated in the table above, vendor extensions are used to configure API-defined base readers, and may also be used by `getLRSpec` to report the configuration of externally-defined base readers. Such vendor extensions MAY be vendor-specific properties that appear in the `properties` parameter of `LRSpec` and may be modified and accessed through the `setProperties` and `getPropertyValue` methods, or they MAY be vendor extensions to `LRSpec` itself, or both.

10.4 LRSpec

An `LRSpec` describes the configuration of a Logical Reader.

LRSpec
`isComposite : Boolean`
`readers : List<String>`
`properties: List<LRProperty>`
`<<extension point>>`
`---`

The ALE implementation SHALL interpret the fields of an `LRSpec` as follows.

Field	Type	Description
`isComposite`	`Boolean`	(Optional) If true, this Logical Reader is a composite reader that is an alias for the logical reader or readers specified in the `readers` field. If false, this Logical Reader is a base reader. Defaults to false if omitted.
`readers`	`List<String>`	(Optional) If `isComposite` is true, an unordered list of zero or more names of logical readers that collectively provide the channel to access Tags represented by this Logical Reader. Specifying the name of this Logical Reader in an ECSpec or CCSpec is equivalent to specifying the names in `readers`, except that different properties may apply. Omitted if `isComposite` is false.
`properties`	`List<LRProperty>`	An unordered list of properties (key/value pairs) that control how Tags are accessed using this Logical Reader.

Table 98. `LRSpec` Fields

The `define` or `update` methods of the Logical Reader API SHALL raise a `ValidationException` under any of the following circumstances:

- `isComposite` is false and `readers` is specified and non-empty.

- `isComposite` is false and the implementation does not support using the Logical Reader API to define base readers.
- `isComposite` is false, the implementation does support using the Logical Reader API to define base readers, but the LRSpec does not conform to the vendor-specific rules for such use.
- `isComposite` is true and any element of `readers` is not a known Logical Reader name.
- A property name in `properties` is not recognized by the implementation.
- The value specified for a property is not a legal value for that property.

10.5 LRProperty

A logical reader property is a name-value pair. Values are generically represented as strings in the Logical Reader API. The ALE implementation is responsible for any data type conversions that may be necessary.

LRProperty
name : String
value : String

The ALE implementation SHALL interpret the fields of an `LRProperty` as follows.

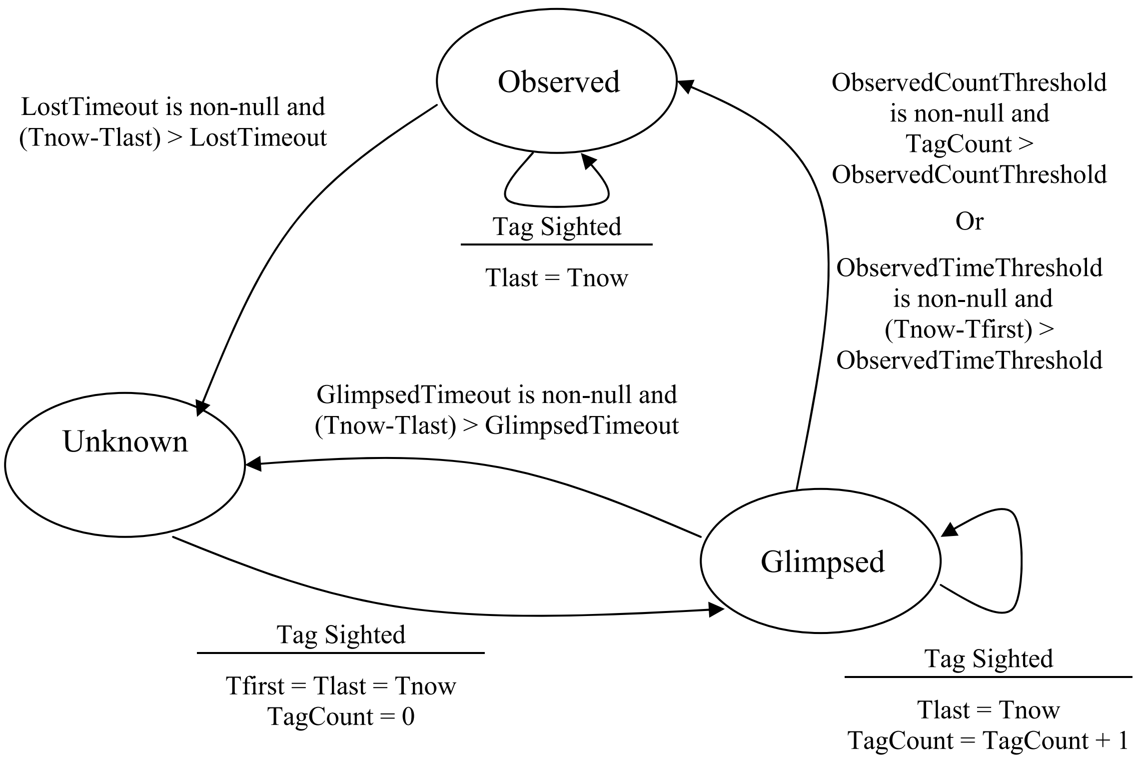
Field	Type	Description
`name`	`String`	The name of the property. The recognized names for properties are implementation-defined. An implementation MAY recognize the standardized properties for tag smoothing defined in Section 10.6
`value`	`String`	(Optional) The value of the property.

Table 99. `LRProperty` Fields

10.6 Tag Smoothing

Tag smoothing is a mechanism whereby a logical reader can be configured to reduce the appearance of tags moving in and out of a reader's field of view due to intermittent tag reads. Smoothing is analogous to circuit-switch debouncing logic. The logic for smoothing is specified by a finite state machine that is evaluated independently for each Tag. Associated with each Tag is the current state (one of the three states in the diagram and table below), real time values Tfirst and Tlast, and a counter TagCount. The real time value Tnow refers to the current time. Events that affect the state machine include the sighting by the logical reader of the Tag and the expiration of certain time intervals calculated by comparing the difference between Tnow and one of the state variables Tfirst or Tlast to a configured timeout threshold.

The finite state machine is illustrated by the following diagram:



The smoothing finite state machine is also specified by the following state table:

State	Event/Condition	Action	Next State
Unknown	Tag Sighted	Tfirst=Tlast=Tnow TagCount=0	Glimpsed
Glimpsed	Tag Sighted	TagCount=TagCount+1 Tlast = Tnow	Glimpsed
Glimpsed	GlimpsedTimeout is non-null and (Tnow-Tlast) > GlimpsedTimeout	--	Unknown
Glimpsed	ObservedTimeThreshold is non-null and (Tnow-Tfirst) > ObservedTimeThreshold	--	Observed
Glimpsed	ObservedCountThreshold is non-null and TagCount > ObservedCountThreshold	--	Observed
Observed	Tag Sighted	Tlast=Tnow	Observed
Observed	LostTimeout is non-null and (Tnow-Tlast) > LostTimeout	--	Unknown

Table 100. Tag Smoothing State Transitions

The application of this smoothing state machine is that, at any point in time, a Reader SHALL consider a Tag to be within view if the Tag is in the Observed state. If an ALE implementation supports smoothing (that is, if an ALE implementation does not raise a `ValidationException` when a client sets the properties defined below), then it SHALL apply the above rule when the reader is used in an ECSpec, and MAY apply the rule when the reader is used in a CCSpec.

State transitions in the smoothing state machine are based upon four parameters, which an ALE client may set using the `properties` parameter of an LRSpec or the `setProperties` method of the Logical Reader API, as specified in Section 10.3. An ALE implementation SHALL interpret these parameters as follows:

Property Name	Description
`GlimpsedTimeout`	A threshold, in milliseconds, that governs the transition between the Glimpsed state and the Unknown state. If a Tag is in the Glimpsed state and is not seen for `GlimpsedTimeout` milliseconds or more, it transitions to the Unknown state. Note that a too-small value for `GlimpsedTimeout`, including a zero value, will prevent a Tag from ever entering the Observed state and therefore prevent any Tag from being operated upon. If `GlimpsedTimeout` is null, a Tag never transitions from the Glimpsed state to the Unknown state.
`ObservedTimeThreshold`	A threshold, in milliseconds, that governs the transition between the Glimpsed state and the Observed state. If a Tag has been in the Glimpsed state for at least `ObservedTimeThreshold`, it transitions to the Observed state. If `ObservedTimeThreshold` is zero, a Tag transitions immediately from the Glimpsed state to the Observed state (that is, it enters the Observed state directly from the Unknown state as soon as the conditions for entering the Glimpsed state are met). If `ObservedTimeThreshold` is null, elapsed time is not used as a criteria for determining when a Tag transitions to the Observed state.
`ObservedCountThreshold`	A threshold that governs the transition between the Glimpsed State and the Observed state. If a Tag has been sighted at least `ObservedCountThreshold` times while in the Glimpsed state, it transitions to the Observed state. If `ObservedCountThreshold` is zero, a Tag transitions immediately from the Glimpsed state to the Observed state (that is, it enters the Observed state directly from the Unknown state as soon as the conditions for entering the Glimpsed state are met). If `ObservedCountThreshold` is null, TagCount is not used as a criteria for determining when a Tag transitions to the Observed state.

Property Name	Description
`LostTimeout`	A threshold, in milliseconds, that governs the transition between the Observed state and the Unknown state. If a Tag is in the Observed state and has not been sighted for at least `LostTimeout` milliseconds, it transitions to the Unknown state. Note that a too-small value for `LostTimeout`, including a zero value, will cause a Tag to transition immediately to the Unknown state from the Observed state. In those cases, however, the implementation SHALL include the Tag in the operation of any active ECSpec or CCSpec. If `LostTimeout` is null, a Tag never transitions from the Observed state to the Unknown state.

Table 101. Tag Smoothing Properties

Notwithstanding the foregoing, if all four properties are set to null for a given logical reader an implementation SHALL NOT use smoothing for that logical reader.

The `define`, `update`, and `setProperties` methods of the Logical Reader API SHALL raise a `ValidationException` under any of the following circumstances:

- If the value of any of the four properties specified above is a non-null string that is not parseable as a non-negative decimal integer numeral.
- If the value of any of the four properties specified above is non-null, and the implementation does not support Tag smoothing for the specified logical reader.
- If both `ObservedTimeThreshold` and `ObservedCountThreshold` are null, and any of the other smoothing parameters is non-null.
- If the implementation does not wish to support the combination of the four parameter values that would result from the operation. An implementation that supports smoothing for the specified logical reader SHALL NOT, however, raise a `ValidationException` for the case where all four parameters are set to null.

11 Access Control API

This section defines an API through which administrative clients can control access by ALE clients to ALE API features. This API provides a standardized, role-based way to associate access control permissions with ALE client identities. The authentication of client identities is binding-specific and outside the scope of this API. The specification of the Access Control API follows the general rules given in Section 4.

The access control model provided by this API is as follows. Each client of the ALE API is presumed to have an identity, authenticated by a binding-specific mechanism. A client identity maps to one or more roles. A role maps to one or more permissions, each of which describes access to a particular feature of the ALE API. The ALE client is permitted to do those things that are described by all of the permissions assigned to all of the roles to which the client identity maps.

Permissions are of two kinds. "Function" permissions grant the right to use a particular method or methods of the ALE API. An example would be a permission that says whether a client is permitted to use the `define` method. In general, if a client attempts an operation that is denied by lack of the appropriate function permission, the operation raises a `SecurityException`. The second kind of permission is a "data" permission, which grants the right to use particular resources or data. An example would be a permission that governs which logical readers a client may use. In general, lack of a data permission does not raise a `SecurityException`, but instead merely limits the data or resources visible through the API.

Permissions are described in the following manner. A "resource" is something within an ALE implementation that a particular client may be granted permission to use. A resource may be a particular API method, or some other resource an implementation wishes to control access to. Each resource is described by a specific class/instance pair. For example, access control for the ALE Writing API is governed by the ALECC.subscribe instance within the APIMethod permission class.

This style of naming resources is extensible, by adding additional class or instance names.

Permissions are described by granting access to specific resources. A client may access only the resources for which at least one permission allows access.

The instance name "`*`" is a wildcard – it means "all instances, including those yet to exist". This is useful in cases where administrators need to grant a wide range of permissions to a client. Because "`*`" is interpreted at the time of the permission check (as opposed to the time of the grant), it accommodates future changes in the underlying configuration.

ALE implementations MAY provide a set of default permissions and roles if they choose.

11.1 API

```
<<interface>>
ALEAC
---
getPermissionNames() : List<String>
definePermission(permName : String, perm : ACPermission) : void
updatePermission(permName : String, perm : ACPermission) : void
getPermission(permName : String) : ACPermission
undefinePermission(permName : String) : void

getRoleNames() : List<String>
defineRole(roleName : String, role : ACRole) : void
updateRole(roleName : String, role : ACRole) : void
getRole(roleName : String) : ACRole
undefineRole(roleName : String) : void
addPermissions(roleName : String, permissionNames : List<String>) : void
setPermissions(roleName : String, permissionNames : List<String>) : void
removePermissions(roleName : String, permissionNames : List<String>) : void

getClientIdentityNames() : List<String>
defineClientIdentity(identityName : String, id : ACClientIdentity) : void
updateClientIdentity(identityName : String, id : ACClientIdentity) : void
getClientIdentity(identityName : String) : ACClientIdentity
getClientPermissionNames(identityName : String) : List<String> // (permission names)
undefineClientIdentity(identityName : String) : void
addRoles(identityName : String, roleNames : List<String>) : void
```

```
removeRoles(identityName : String, roleNames :
List<String>) : void
setRoles(identityName : String, roleNames : List<String>) :
void

getSupportedOperations() : List<String>
getStandardVersion() : String
getVendorVersion() : String
<<extension point>>
```

An ALE implementation SHALL implement the methods of the ALE Access Control API as specified in the following table:

Method	Description
`getPermissionNames`	Returns an unordered list of the names of all permissions.
`definePermission`	Creates a new permission named `permName` according to the specified `perm`.
`updatePermission`	Changes the definition of the permission named `permName` to match the specification in the `perm` parameter. This is different than calling `undefinePermission` followed by `definePermission`, because `updatePermission` may be called even if there are defined roles that refer to this permission.
`undefinePermission`	Removes the permission named `permName`.
`getPermission`	Returns an `ACPermission` that describes the permission named `permName`.
`getRoleNames`	Returns an unordered list of the names of all roles.
`defineRole`	Creates a new role named `roleName` according to the specified `role`.
`updateRole`	Changes the definition of the role named `roleName` to match the specification in the `role` parameter. This is different than calling `undefineRole` followed by `defineRole`, because `updateRole` may be called even if there are defined client identities that refer to this role.
`undefineRole`	Removes the role named `roleName`.

Method	Description
`getRole`	Returns an `ACRole` that describes the role named `roleName`.
`addPermissions`	Adds the specified permissions to the list of permissions for the role named `roleName`. This is equivalent to calling `getRole`, modifying the `ACRole` that is returned to include the specified permissions in the list of permission names, and then calling `updateRole` with the modified `ACRole`.
`setPermissions`	Changes the list of permissions for the role named `roleName` to the specified list. This is equivalent to calling `getRole`, modifying the `ACRole` that is returned by replacing the permission list with the specified list of permissions, and then calling `updateRole` with the modified `ACRole`.
`removePermissions`	Removes the specified permissions from the list of permissions for the role named `roleName`. Any permission name within `perms` that is not currently among the permissions of the specified role is ignored. This is equivalent to calling `getRole`, modifying the `ACRole` that is returned by removing any references to permissions in the specified permission list, and then calling `updateRole` with the modified `ACRole`.
`getClientIdentityNames`	Returns an unordered list of the names of all client identities.
`defineClientIdentity`	Creates a new client identity named `identityName` according to the specified `ClientIdentity`.
`updateClientIdentity`	Changes the definition of the client identity named `identityName` to match the specification in the `ClientIdentity` parameter. This is different than calling `undefineClientIdentity` followed by `defineClientIdentity`, because `updateClientIdentity` may be called even if there are defined client identities that refer to this client identity.
`undefineClientIdentity`	Removes the client identity named `identityName`.

Method	Description
`getClientIdentity`	Returns an `ACClientIdentity` that describes the client identity named `identityName`.
`addRoles`	Adds the specified roles to the list of roles for the client identity named `identityName`. This is equivalent to calling `getClientIdentity`, modifying the `ACClientIdentity` that is returned to include the specified roles in the list of role names, and then calling `updateClientIdentity` with the modified `ACClientIdentity`.
`setRoles`	Changes the list of roles for the client identity named `identityName` to the specified list. This is equivalent to calling `getClientIdentity`, modifying the `ACClientIdentity` that is returned by replacing the role list with the specified list of roles, and then calling `updateClientIdentity` with the modified `ACClientIdentity`.
`removeRoles`	Removes the specified roles from the list of roles for the client identity named `identityName`. Any role name within `perms` that is not currently among the roles of the specified client identity is ignored. This is equivalent to calling `getClientIdentity`, modifying the `ACClientIdentity` that is returned by removing any references to roles in the specified role list, and then calling `updateClientIdentity` with the modified `ACClientIdentity`.
`getClientPermissionNames`	Returns an unordered list of all permission names granted to the specified client identity. This is equivalent to calling `getRoles`, then combining the results of calling `getPermissions` for each role listed in the result from `getRoles`.
`getSupportedOperations`	Returns an unordered list of all methods within the Access Control API that are implemented by the implementation (that is, those methods that do not raise `UnsupportedOperationException`). See Section 11.8.

Method	Description
`getStandardVersion`	Returns a string that identifies what version of the specification this implementation of the ALE Access Control API complies with, as specified in Section 4.3.
`getVendorVersion`	Returns a string that identifies what vendor extensions of the ALE Access Control API this implementation provides, as specified in Section 4.3.

Table 102. ALEAC Interface Methods

11.2 Error Conditions

Methods of the Access Control 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 implementation SHALL raise this exception if the client was not granted access rights to the called method. Other, implementation-specific circumstances may cause this exception; these are outside the scope of this specification.
`NoSuchPermissionException`	The specified permission name doesn't exist.
`PermissionValidationException`	The specified permission is invalid according to Section 11.6, or for the `definePermission` method the specified `permissionName` is an empty string or is not accepted by the implementation according to Section 4.5.
`DuplicatePermissionException`	There already exists a permission having the specified name.
`NoSuchRoleException`	The specified role name doesn't exist.

Exception Name	Meaning
`RoleValidationException`	The specified role is invalid according to Section 11.5, or for the `defineRole` method the specified `roleName` is an empty string or is not accepted by the implementation according to Section 4.5.
`DuplicateRoleException`	There already exists a role having the specified name.
`NoSuchClientIdentityException`	The specified client identity name doesn't exist.
`ClientIdentityValidationException`	The specified client identity is invalid according to Section 11.3, or for the `defineClientIdentity` method the specified `clientIdentityName` is an empty string or is not accepted by the implementation according to Section 4.5.
`DuplicateClientIdentityException`	There already exists a client identity having the specified name.
`UnsupportedOperationException`	The implementation does not provide this method. See Section 11.8.
`ImplementationException`	A generic exception raised 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.

Table 103. Exceptions in the ALEAC Interface

The exceptions that may be raised by each Access Control API method are indicated in the table below. An ALE implementation SHALL raise the appropriate exception listed below when the corresponding condition described above occurs. If more than one exception condition applies to a given method call, the ALE implementation may raise any of the exceptions that applies.

ALE Method	Exceptions
getPermissionNames	UnsupportedOperationException SecurityException ImplementationException
definePermission	SecurityException DuplicatePermissionException PermissionValidationException UnsupportedOperationException ImplementationException
updatePermission	NoSuchPermissionException PermissionValidationException UnsupportedOperationException SecurityException ImplementationException
getPermission	SecurityException NoSuchPermissionException UnsupportedOperationException ImplementationException
undefinePermission	SecurityException NoSuchPermissionException UnsupportedOperationException ImplementationException
getRoleNames	SecurityException UnsupportedOperationException ImplementationException
defineRole	SecurityException DuplicateRoleException RoleValidationException UnsupportedOperationException ImplementationException
updateRole	NoSuchRoleException RoleValidationException UnsupportedOperationException SecurityException ImplementationException
getRole	SecurityException NoSuchRoleException UnsupportedOperationException ImplementationException

ALE Method	Exceptions
undefineRole	SecurityException NoSuchRoleException UnsupportedOperationException ImplementationException
addPermissions	SecurityException NoSuchRoleException NoSuchPermissionException UnsupportedOperationException ImplementationException
setPermissions	SecurityException NoSuchRoleException NoSuchPermissionException UnsupportedOperationException ImplementationException
removePermissions	SecurityException NoSuchRoleException UnsupportedOperationException ImplementationException
getClientIdentityNames	UnsupportedOperationException SecurityException ImplementationException
defineClientIdentity	SecurityException DuplicateClientIdentityException ClientIdentityValidationException UnsupportedOperationException ImplementationException
updateClientIdentity	SecurityException NoSuchClientIdentityException ClientIdentityValidationException UnsupportedOperationException ImplementationException
getClientIdentity	SecurityException NoSuchClientIdentityException UnsupportedOperationException ImplementationException
getClientPermissionNames	SecurityException NoSuchClientIdentityException UnsupportedOperationException ImplementationException

ALE Method	Exceptions

`undefineClientIdentity`	`SecurityException`
	`NoSuchClientIdentityException`
	`UnsupportedOperationException`
	`ImplementationException`

`addRoles`	`SecurityException`
	`NoSuchClientIdentityException`
	`NoSuchRoleException`
	`UnsupportedOperationException`
	`ImplementationException`

`removeRoles`	`SecurityException`
	`NoSuchClientIdentityException`
	`UnsupportedOperationException`
	`ImplementationException`

`setRoles`	`SecurityException`
	`NoSuchClientIdentityException`
	`NoSuchRoleException`
	`UnsupportedOperationException`
	`ImplementationException`

`getSupportedOperations`	`ImplementationException`
`getStandardVersion`	`ImplementationException`
`getVendorVersion`	`ImplementationException`

Table 104. Exceptions Raised by each ALEAC Interface Method

11.3 ACClientIdentity

An `ACClientIdentity` identifies a client that may access the ALE API.

```
ACClientIdentity
credentials : List<ACClientCredential>
roleNames : List<String> // list of role names
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of an `ACClientIdentity` as follows.

Field	Type	Description
credentials	List<ACClientCredential>	An unordered list of zero or more credentials that the implementation may use to authenticate the identity of this client.
roleNames	List<String>	An unordered list of the names of zero or more roles that are assigned to this client identity.

Table 105. ACClientIdentity Fields

The defineClientIdentity, and updateClientIdentity methods of the Access Control API SHALL raise a ClientIdentityValidationException under any of the following circumstances:

- One or more of the specified credentials is not a valid credential, according to the implementation-specific rules for validating credentials.
- One or more of the specified roleNames is not a known name for a role.

11.4 ACClientCredential

An ACClientCredential is information that the ALE implementation uses to authenticate the identity of an API client. The contents of a credential and how it is used in the authentication process is implementation specific, and hence this type is defined as purely an extension point.

```
ACClientCredential
<<extension point>>
---
```

11.5 ACRole

An ACRole describes a role that may be assigned to a client identity.

```
ACRole
permissionNames  : List<String> // of permission names
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of an ACRole as follows.

Field	Type	Description

Field	Type	Description
`permissionNames`	`List<String>`	An unordered list of the names of zero or more permissions that are granted to all client identities to which this role is assigned.

Table 106. `ACRole` Fields

The `defineRole`, and `updateRole` methods of the Access Control API SHALL raise a `RoleValidationException` under any of the following circumstances:

- One or more of the specified `permissionNames` is not a known name for a permission.

11.6 ACPermission

An `ACPermission` describes one or more specific permissions that may be associated with a role and thereby granted to client identities.

```
ACPermission
permissionClass : ACClass
instances : List<String>
<<extension point>>
---
```

The ALE implementation SHALL interpret the fields of an `ACPermission` as follows.

Field	Type	Description
`permissionClass`	`ACClass`	The permission class within which the names in `instances` are to be interpreted. See Section 11.7.
`instances`	`List<String>`	An unordered list of one or more instances of the specified permission class. This permission grants permission to use all of the instances within the specified class that are specified in this list. See Section 11.7.

Table 107. `ACPermission` Fields

The `definePermission`, and `updatePermission` methods of the Access Control API SHALL raise a `PermissionValidationException` under any of the following circumstances:

- The specified `permissionClass` is not a known permission class.
- One or more of the specified `instances` is not a valid instance string for the specified permission class, according to the table in Section 11.7.

11.7 Access Permission Classes (ACClass)

An `ACClass` is an extensible, enumerated type denoting a permission class.

<<Enumerated Type>> ACClass
`Method`
<<extension point>>

An ALE implementation SHALL recognize the following permission class names, and implement each according to the following table.

Permission Class	Description	Valid Instances
`Method`	Each instance specifies an API method or a set of API methods to which permission is granted. If a client has not been granted permission for a given method, if that client calls the method the ALE implementation SHALL raise a `SecurityException`. However, an ALE implementation SHALL NOT raise a `SecurityException` for a method whose specification does not include `SecurityException` as a possible error condition, regardless of permission settings. This includes the `getStandardVersion` and `getVendorVersion` methods of all ALE APIs, and the `getSupportedOperations` method of the Access Control API.	The name of a specific method, an API name, or the asterisk character (`*`). Specifying the name of an API grants permission to all methods of the API, including any vendor extension methods. Specifying the asterisk character grants permission to all methods of all APIs. See Section 11.7.1.

Table 108. `ACClass` Values

11.7.1 Instance Names for the Method Class

An instance for the `Method` permission class is either the name of a specific method, the name of an API, or a wildcard (`*`). An ALE implementation SHALL recognize the following strings as API names when they appear as instances for the `Method` permission class, denoting that permission is granted to use all methods of the specified API, including vendor extensions.

Instance Name	Description
`ALE`	ALE Reading API

Instance Name	Description
ALECC	ALE Writing API
ALETM	ALE Tag Memory API
ALELR	ALE Logical Reader API
ALEAC	ALE Access Control API

Table 109. Method Permission Class Instance Names for APIs

A specific method is indicated by an instance name consisting of an API name as defined above, a period (.), and the name of a method within that API. An ALE implementation SHALL recognize a string of that form as a method name when it appears as an instance for the Method permission class. For example, the string ALECC.subscribe denotes the subscribe method of the ALE Writing API.

An ALE implementation SHALL recognize the string consisting of a single asterisk character (*) as denoting all methods of all APIs when it appears as an instance for the Method permission class.

11.8 Partial Implementations

An implementation of the Access Control API SHALL implement all methods as specified in Section 11.1. Unlike other ALE APIs, however, the Access Control API specifies facilities that may overlap or conflict with facilities provided by the environment in which other ALE APIs are provided. For example, it is common in large enterprises to centralize information about identities, roles, and permissions in repositories such as LDAP servers, so that this information may be shared across many different applications. In such a setting, it may not be appropriate for the system component including an ALE implementation to provide its own API for manipulating client identities and permissions, but instead defer to the mechanisms provided by the LDAP environment.

For this reason, most methods of the Access Control API can raise an UnsupportedOperationException. An ALE implementation MAY raise UnsupportedOperationException from an Access Control API rather than carrying out the normal function of the method, if the implementation does not wish to provide that feature through the ALE Access Control API. If an implementation raises UnsupportedOperationException from any Access Control API method, it SHALL provide documentation that specifies how the client or user controls components of the access control model – client identities, roles, and permissions – for which Access Control API methods raise the UnsupportedOperationException. For example, an implementation may specify that client identities, identity-to-role mappings, and role-to-permission mappings are obtained from an external LDAP server, and that permissions are defined and manipulated using Access Control API. In that example, all of the methods concerned with defining and manipulating client identities and roles might raise UnsupportedOperationException.

Explanation (non-normative): It is common in large enterprises to centralize information about identities, roles, and permissions in repositories such as LDAP servers, so that this information may be shared across many different applications. The provisions of this section are specifically intended to allow ALE implementations to work in such an environment. In addition, the reason that permission names are introduced in the API is so that client-to-role and role-to-permission mappings may be stored externally in a way that only requires storing strings, as opposed to more complex objects, in the external repository. At the same time, there may also be ALE implementations that wish to provide a fully-featured access control system that is self-contained, that is, that does not require an external repository. The Access Control API includes a full set of methods for manipulating client identities, roles, and permissions so that self-contained access control implementations will have a standardized interface.

In order to insure that implementations provide a reasonable set of facilities to clients, an ALE implementation SHALL conform to the following rules for selecting which methods raise `UnsupportedOperationException`. In these rules, to "support" a method means to implement the method according to the preceding sections, and never to raise `UnsupportedOperationException`. Conversely, to not support a method means to always raise `UnsupportedOperationException`.

1. An implementation SHALL always support `getStandardVersion`, `getVendorVersion`, and `getSupportedOperations`.
2. For methods related to permissions, an implementation SHALL choose one of the following four alternatives:
    - 2.1. No methods supported.
    - 2.2. Support only `getPermissionNames`.
    - 2.3. Support `getPermissionNames`, `definePermission`, `undefinePermission`, and `getPermission`.
    - 2.4. Support all of the methods in 2.3, plus `updatePermission`.
3. For methods related to roles, an implementation SHALL choose one of the following four alternatives:
    - 3.1. No methods supported.
    - 3.2. Support only `getRoleNames`.
    - 3.3. Support `getRoleNames`, `defineRole`, `undefineRole`, and `getRole`.
    - 3.4. Support all of the methods in 3.3, plus `updateRole`, `addPermissions`, `setPermissions`, and `removePermissions`.
4. For methods related to client identities, an implementation SHALL choose one of the following four alternatives:
    - 4.1. No methods supported.
    - 4.2. Support only `getClientIdentityNames`.

4.3. Support `getClientIdentityNames`, `defineClientIdentity`, `undefineClientIdentity`, and `getClientIdentity`.

4.4. Support all of the methods in 4.3, plus `updateClientIdentity`, `addRoles`, `setRoles`, and `removeRoles`.

5. If an implementation supports `getClientIdentity` and `getRole`, it SHALL also support `getClientPermissionNames`.

The `getSupportedOperations` method is provided so that clients may easily determine which methods are supported and which are not. As a consequence of the above rules, the list returned by `getSupportedOperations` SHALL always include the strings `getStandardVersion`, `getVendorVersion`, and `getSupportedOperations` (and possibly others).

11.9 Anonymous User

An implementation MAY allow clients to access one or more ALE APIs without authenticating the client identity, either by using a binding that does not support authentication or by omitting the authentication step in a binding that does. If an implementation does provide unauthenticated access, the implementation SHOULD provide a special "anonymous" client identity that can be used to control the access rights of an unauthenticated client. For example, an implementation may use the special string "`<anonymous>`" to denote the anonymous client identity, and then unauthenticated clients will be granted access according to what roles and permissions are assigned to client identity `<anonymous>`. An implementation SHALL provide documentation to specify whether an anonymous client identity is provided, and if so what its name is.

11.10 Initial State

In order to grant access to ordinary clients, there must exist at least one client who has permission to use the Access Control API, or there must be some out-of-band mechanism for establishing access permissions. An implementation SHALL provide documentation that specifies how this is done.

Example (non-normative): If an ALE implementation's sole means to configure access permissions is through the ALE Access Control API, then the implementation might provide an initial "superuser" client identity that is initially granted permission for everything. The client identity name and credentials for this initial "superuser" might be configurable at product installation time, or might be a fixed string and password. If an ALE implementation uses an external source of client identities as described in Section 11.8, then it may be sufficient simply to rely on whatever means that external system provides for configuring resources.

12 Use Cases (non-normative)

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

12.1 Reading API Use Cases

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 event 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 event 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 event 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)
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)

Table 110. Summary of ALE Interface Use Cases

12.2 Writing API Use Cases

1. A **high speed conveyor carries cases of a product**, where each case **contains a dozen innerpacks**. Both the cases and the innerpacks contain tags whose EPC memory is already programmed in a way that the case and innerpacks can be distinguished (e.g., through the "filter" bits). As each case passes a reader, the ALE implementation is to write a lot code into the user memory of the innerpacks only.

   The above use case explores the need to have write command be applied selectively to tags based on filtering. The high speed aspect is intended to illustrate the need to give an implementation the freedom to carry out the intent within a single or a small number of Gen2 "inventory rounds".

2. A **high speed conveyor carries cases of a product**, where each case **contains 24 items of identical product**. The case has a tag that has been pre-programmed with an SGTIN derived from a known GTIN. Each item carries a tag whose EPC memory is not yet written. As each case passes a reader, the ALE implementation is to assign 24 unique EPCs based on the GTIN and ensure each item has a unique EPC written to its tag.

   The above use case explores the need to assign EPCs when many tags are within view of the reader, and without tag-by-tag intervention of the ALE client.

3. **Same as the prior use case, but a unique kill password** is also to be assigned to each item.

   The above use case explores the need to assign unique kill passwords, perhaps based on generating random numbers, without tag-by-tag intervention of the ALE client.

4. At a **retail checkout location**, the ALE implementation is to kill all tags (or a designated subset of tags) within view of a designated set of readers. Each tag has a distinct kill password, and the mapping of EPCs to kill passwords for all items that might arrive at checkout is known in advance.

   This use case explores the need to do associative lookup to determine kill passwords to use.

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

Writing API Use Case	**Command Cycle Boundaries**	**Command Spec**				**Report Content**
		Filter	**Operation Spec(s)**			
			Op Type	**Field Spec**	**Data Spec**	
1. High Speed Conveyor writing lot code to inner packs.	Triggered by case entering and leaving the reader tunnel	`INCLUDE` Inner packs (based on its "filter" bits)	`WRITE`	Lot field	`LITERAL` with the specified lot code	A list of 12 CCTagReport instances, one for each inner pack. Fewer than 12 indicates a problem.
2. High Speed Conveyor writing EPCs	Triggered by case entering and leaving read tunnel	`EXCLUDE` Case (based on its SGTIN or SGTIN pattern for its GTIN)	`WRITE`	EPC field	`CACHE` with the specified EPC cache	A list of 24 CCTagReport instances, each giving the specific EPC value written for one item
3. Assignment of Kill Password	Triggered by case entering and leaving read	`EXCLUDE` Case (based on its SGTIN or SGTIN	`WRITE`	EPC field	`CACHE` with specified EPC cache	A list of 24 CCTagReport instances, each giving the specific

Writing API Use Case	Command Cycle Boundaries	Command Spec				Report Content
		Filter	Operation Spec(s)			
			Op Type	Field Spec	Data Spec	
	tunnel	pattern for its GTIN)	WRITE	KillPwd	RANDOM	EPC value and corresponding kill password written for one item
4. Kill tags at Retail Checkout	Triggered by item entering kill zone, or by manual signal from checkout clerk	Specific items checked out	KILL		ASSO-CIATION with a table mapping EPCs to kill passwords	A list of CCTagReport instances for each item, indicating successful kill. The number of instances can be compared to the number of items checked out to detect problems.

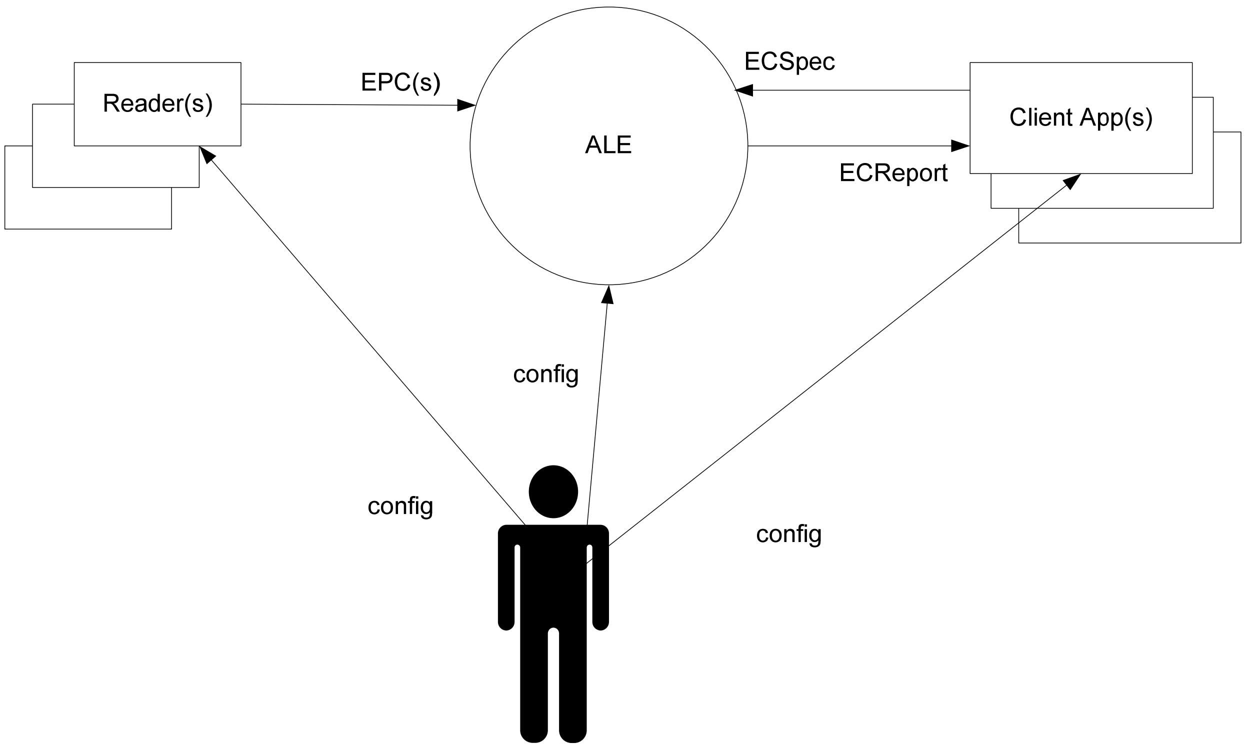
Table 111. Summary of ALECC Interface Use Cases

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. The illustration is based on the Reading API, but the API-level interaction patterns are identical for the Writing API.

13.1 ALE Context

An ALE implementation exists in a context including RFID readers or other devices, Users (administrative) and Client applications as shown below. Initially the administrators are responsible for installing and configuring the environment. Once the environment is configured, Tag data are sent from the Readers to the ALE implementation. In some cases the ALE implementation may be embedded in a reader device, but for clarity the illustrations below show the Reader as a separate component



The ALE clients are applications or services that interact with the ALE implementation. 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`

These methods are defined normatively for the Reading API in Section 8.1. The Writing API has corresponding methods, defined normatively in Section 9.1.

13.2 Interaction Scenarios

Three sequence diagrams are illustrated below to demonstrate the use of the ALE Reading or Writing API. The three sequence diagrams correspond to three ways a client may cause event cycles or command cycles to occur:

1. Subscribing to a previously defined ECSpec (or CCSpec) in order to receive asynchronous notifications via the callback interface.

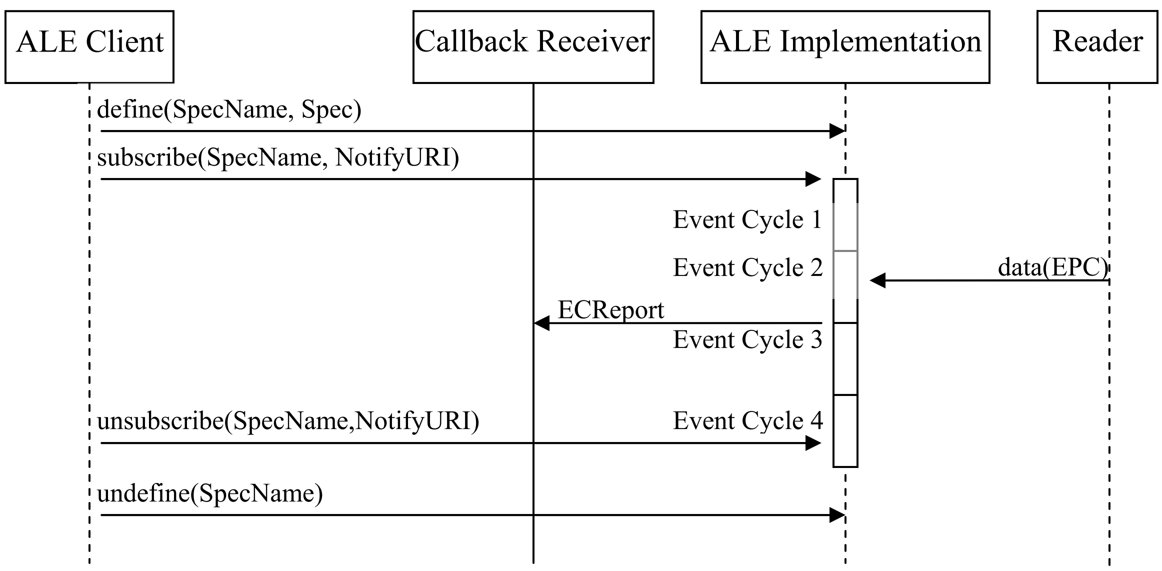
2. Polling a previously defined ECSpec or CCSpec in order to receive a synchronous result.

3. Using the immediate method to present a single-use ECSpec or CCSpec in order to receive a synchronous result.

In each of the sequence diagrams below, interactions between the ALE Implementation and a Reader are depicted. The ALE specification is purposefully silent on how a Reader communicates with an ALE Implementation and does not favor any mechanism in particular. Likewise the specification is purposefully silent on how an ALE Implementation and a Reader coordinate with each other. Therefore, the diagrams generically show the Reader / ALE Implementation interaction as a single arrow from Reader to ALE Implementation labeled "data(EPC)." This is not meant to suggest that the Reader / ALE Implementation interaction is always a "push" of data from Reader to ALE Implementation, nor that an ALE Implementation must have a method called "data." The "data(EPC)" arrow is merely a placeholder for whatever implementation-specific mechanism is used.

13.2.1 Subscribing for Asynchronous Notifications

This scenario illustrates the interaction between different entities in the context of a subscription for aynchronous notification of reports.



13.2.1.1 Assumptions

- All configuration, and initialization required has already been performed.
- The *ALE Implementation* implements ALE API.
- The *ALE Client* is the only subscriber

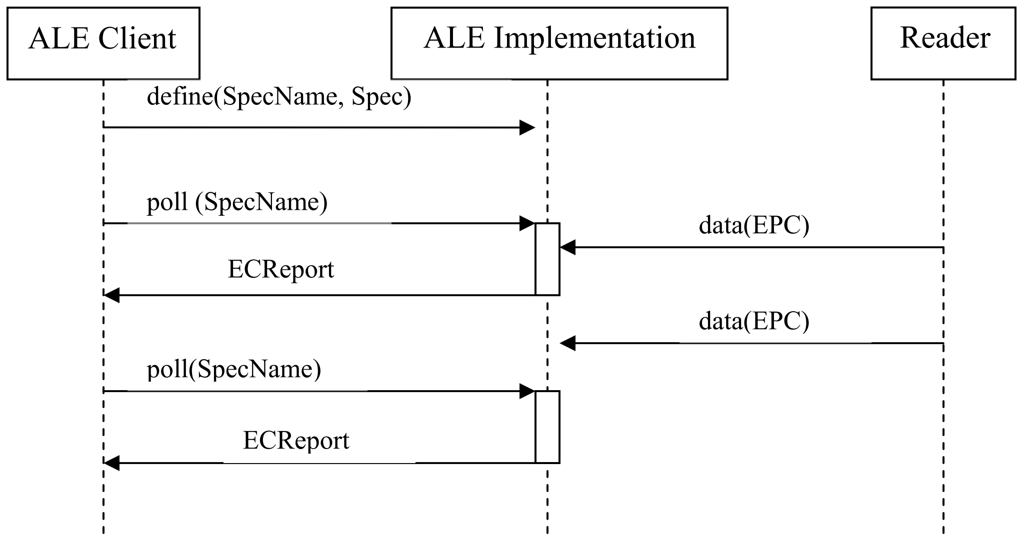
- No filtering is performed by ECSpecs.
- No tag smoothing is being performed.
- The *Callback Receiver* receives reports at NotifyURI.
- The interaction of the Reader with the ALE Implementation is indicated by the "data(EPC)" arrow, as explained earlier.
- This is a normal scenario involving no errors.

13.2.1.2 Description

1. The *ALE Client* calls the `define` method of the ALE interface. The ECSpec specifies a repeat period of zero (implying that an event cycle begins as soon as the previous one ends), and a duration of five seconds. The ECSpec includes a single ECReportSpec wherein the `reportSet` is set to ADDITIONS, and `reportIfEmpty` is set to false. At this point the ECSpec is considered "Unrequested."
2. The client calls the `subscribe` method, including a URI that identifies the *Callback Receiver* as the destination for the `ECReports`. In this scenario, the callback receiver is shown as a separate entity receiving ECReports. In some instances, the client could be the callback receiver. 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 Event Cycle 1 no new tags (additions) were reported by the Reader so no `ECReports` is generated.
4. In Event Cycle 2, an EPC is reported to the ALE Implemetnation by one of the Readers indicated in the ECSpec.
5. At the end of event Cycle 2, an `ECReports` instance is generated and sent to the client.
6. In Event Cycle 3, no EPCs are reported by the Reader, and no `ECReports` are generated.
7. In Event Cycle 4 the client calls the `unsubscribe` method of the ALE interface. As this removes the only subscriber, the ECSpec transitions to the "Unrequested" state, and no further reads are performed nor `ECReports` generated.
8. Finally, the ALE Client calls `undefine` method of ALE interface to remove the ECSpec from the *ALE Implementation.*

13.2.2 Polling for Synchronous Results

This scenario illustrates the interaction between different entities in the context of a polling request.



13.2.2.1 Assumptions

- All configuration, and initialization required have already been performed.
- The *ALE Implementation* implements ALE API.
- The *ALE Client* is the only client requesting reports from the server.
- No filtering is performed by ECSpecs.
- No tag smoothing is being performed.
- The interaction of the Reader with the ALE Implementation is indicated by the "data(EPC)" arrow, as explained earlier.
- This is a normal scenario involving no errors.

13.2.2.2 Description

1. The *ALE Client* calls the `define` method of the ALE interface. The ECSpec specifies a repeat period of zero (implying that one event cycle begins as soon as the previous one ends), and a duration of five seconds. The ECSpec includes a single ECReportSpec wherein the `reportSet` is set to ADDITIONS, and `reportIfEmpty` is set to false. 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.

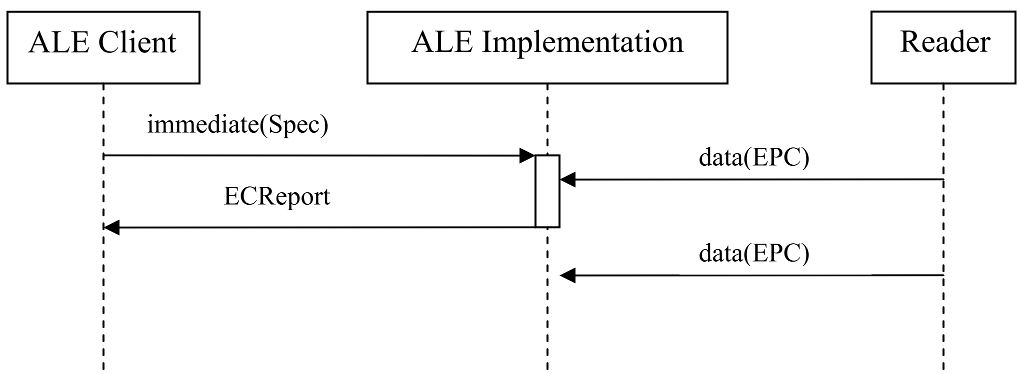
3. An EPC is received during the event cycle. At the end of the event cycle, the `ECReports` is generated and returned to the ALE Client as the response to the `poll` method. At this point the `ECSpec` transitions to the “Unrequested” state.

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 be ignored.

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, an empty `ECReports` instance is returned.

13.2.3 Defining a Single-Use Spec and Receiving a Synchronous Report

This scenario illustrates the interaction between different entities in the context of an `immediate` request.



13.2.3.1 Assumptions

- All configuration, and initialization required has already been performed.
- The *ALE Implementation* implements ALE API.
- The *ALE Client* is the only client requesting reports from the server.
- No filtering is performed by ECSpecs.
- No tag smoothing is being performed.
- The interaction of the Reader with the ALE Implementation is indicated by the “data(EPC)” arrow, as explained earlier.
- This is a normal scenario involving no errors.

13.2.3.2 Description

1. The *ALE Client* calls the `immediate` method of the ALE interface. The ECSpec specifies a repeat period of zero (implying that one event cycle begins as soon as the previous one ends), and a duration of five seconds. The ECSpec includes a single

ECReportSpec wherein the `reportSet` is set to ADDITIONS, and `reportIfEmpty` is set to false. The spec immediately transitions into “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 `immediate` method.

2. An EPC 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 `immediate` method. At this point the ECSpec is removed from the *ALE Implementation*.

14 Appendix: EPC Patterns (non-normative)

EPC Patterns are used to specify filters within `ECFilterSpec` and `CCFilterSpec` instances. The normative specification of EPC Patterns may be found in the EPCglobal Tag Data Specification Version 1.3 [TDS1.3.1]. 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 General Manager Number (*`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

Table 112. EPC Pattern Examples

15 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	A set of interfaces through which ALE Clients may interact with filtered, consolidated EPC data and related data from a variety of sources. In all, there are five APIs and two callback interfaces.
ALE Client	2	A system component, typically application business logic, that interacts with EPC data and related data through an ALE Interface.
ALE Implementation	2	Software or hardware that receives requests from one or more ALE Clients and carries out operations according to this specification.
Access Control API	4, 11	An API through which clients may define the access rights of other clients to use the facilities provided by the other APIs. One of five APIs comprising the ALE Interface.
Callback Interface	4.2, 8.4, 9.8	An interface through which the Reading API and Writing API deliver asynchronous results from standing requests.
CCReports	5.3, 9.4	A command cycle reports instance (CCReports) describes the result of completing a single command cycle. It is provided as an output by an implementation of the ALE Writing API.
CCSpec	5.3, 9.3	A command cycle specification (CCSpec) specifies the operations to be performed by an ALE implementation during a Command Cycle. An ALE Client provides a CCSpec to an ALE Implementation to control the operation of the Writing API.
Command Cycle	5.3	The smallest unit of interaction between an ALE client and an implementation of the ALE Writing API. A command cycle is an interval of time during which Tags are written or otherwise operated upon.

Term	Section	Meaning
Datatype	5.4	Specifies what kind of data values a Tag field is considered to contain, and how they are encoded into the Tag's memory.
ECReports	5.2, 8.3	An event cycle reports instance (ECReports) describes the result of completing a single event cycle. It is provided as an output by an implementation of the ALE Reading API.
ECSpec	5.2, 8.2	An event cycle specification (ECSpec) specifies the operations to be performed by an ALE implementation during an Event Cycle, and how the results are to be reported. An ALE Client provides an ECSpec to an ALE Implementation to control the operation of the Reading API.
Event Cycle	5.2	The smallest unit of interaction between an ALE client and an implementation of the ALE Reading API. An event cycle is an interval of time during which Tags are read.
Fieldname	5.4	A name that specifies a particular data field of a Tag.
Fieldspec	5.4	A structure that is used to specify how a data field of a Tag is accessed through the ALE Interface, consisting of a Fieldname, Datatype, and Format.
Fixed Field	5.4	A Tag memory field that occupies a fixed location. By definition, a fixed field always exists as long as the memory bank exists and is of sufficient size.
Format	5.4	Specifies the syntax by which individual data values are presented at the level of the ALE Interface.
Grouping Operator	5.2.1, 6.2.1.4, 6.2.2.4	A function that maps a data value into a group code. Specifies how data read within an Event Cycle are to be partitioned into groups for reporting purposes.
Logical Reader API	10	An API through which clients may define logical reader names for use with the Reading API and the Writing API, each of which maps to one or more sources/actuators provided by the implementation.
Logical Reader Name	10	An abstract name that an ALE Client uses to refer to one or more Readers that have a single logical purpose; *e.g.*, DockDoor42.
On-demand ("Pull") Request	5.2, 5.3	A request for the execution of an event or command cycle which is carried out on a one-time basis at the time of request. On-demand requests are made using the `immediate` or `poll` methods of the ALE Reading or Writing API. Results are returned directly to the caller at the completion of the event or command cycle.

Term	Section	Meaning
Physical Reader	10	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 Implementation.
Reader	5.1	A channel through which Tags are accessed. Through a Reader, data may be read from Tags, and in some cases (depending on the capabilities of the Readers and Tags involved) data may be written to Tags or other operations performed on Tags.
Reader Cycle	5.1	The smallest unit of interaction of an ALE Implementation with a Reader.
Reading API	5, 6, 8	An API through which clients may obtain filtered, consolidated EPC and other data from a variety of sources. In particular, clients may read RFID tags using RFID readers. One of five APIs comprising the ALE Interface.
Report	5.1	Data about event cycle communicated from the ALE Implementation to an ALE Client.
Standing ("Push") Request	5.2, 5.3	A request for the execution of event or command cycles that remains in effect until subsequently cancelled. During the time the request remains in effect, multiple event or command cycles may be completed. Each time an event or command cycle completes, results are sent asynchronously to one or more Subscribers via the ALECallback or ALECCCallback Interface. Standing requests are entered using the `subscribe` method of the ALE Reading or Writing API, and cancelled using the `unsubscribe` method.
Subscriber	4.2, 8, 9	A receiver of asynchronous results generated from a Standing Request.
Tag	5.1	A data carrier such as an RFID tag or some other data carrier that can be treated in a similar manner such as a bar code, OCR text, and so on.
Tag Memory API	7	An API through which clients may define symbolic names that refer to data fields of tags. One of five APIs comprising the ALE Interface.
Variable Field	5.4	A Tag memory field that does not occupy a fixed location or that may be absent. A variable field may or may not exist depending on the contents of memory. Accessing a variable field may require the presence of additional information to be present in Tag memory locations other than field itself.

Term	Section	Meaning
Writing API	5, 6, 9	An API through which clients may cause operations to be performed on EPC data carriers through a variety of actuators. In particular, clients may write RFID tags using RFID "readers" (capable of writing tags) and printers. One of five APIs comprising the ALE Interface.

Table 113. Glossary

16 Appendix: Changes in ALE 1.1 (non-normative)

This section summarizes the changes between ALE 1.0 and ALE 1.1.

16.1 Changes to the ALE Reading API

- `primaryKeyFields` parameter added to `ECSpec` (Section 8.2).
- More than one start trigger may be specified in an ECSpec. The `startTrigger` parameter of `ECBoundarySpec` is deprecated in favor of a new parameter `startTriggerList`. (Section 8.2.1)
- Start conditions are no longer mutually exclusive: an ECSpec may specify both start triggers and repeat period. (Section 8.2.1)
- More than one stop trigger may be specified. The `stopTrigger` parameter of `ECBoundarySpec` is deprecated in favor of a new parameter `stopTriggerList`. (Section 8.2.1)
- A new stop condition "when data available" is added, indicated by boolean `whenDataAvailable` in `ECBoundarySpec`. (Section 8.2.1)
- A new real-time clock standardized trigger is added. (Section 8.2.4.1)
- A facility for reporting per-reader, per-tag, and per-tag-sighting "statistics" (that is, information beyond the data read from the tag) is added. See the new `statProfileNames` parameter of `ECReportSpec` (Section 8.2.5) and Sections 8.2.13, 8.3.9, 8.3.10, 8.3.11, and 8.3.12.
- Filters have been extended to allow for filtering on any combination of Tag fields. The `includePatterns` and `excludePatterns` parameters of `ECFilterSpec` are deprecated in favor of a new `filterList` parameter. (Sections 8.2.7 and 8.2.8)
- Grouping has been extended to allow for grouping on any single Tag field. (Section 8.2.9)
- `ECReportOutputSpec` has been extended to allow reading of any combination of Tag fields. See the new `fieldList` parameter of `ECReportOutputSpec` (Section 8.2.10) and Sections 8.3.6 and 8.3.7.

- ECReports includes a new parameter initiationCondition to indicate which of several start conditions actually initiated an event cycle, and new fields initiationTrigger and terminationTrigger to indicate which of several triggers were used in the case of initiation or termination via trigger. (Sections 8.3 and 8.3.1)
- New values for ECTerminationCondition added: DATA_AVAILABLE and UNDEFINE. (Section 8.3.2)

16.2 New APIs

The following APIs are completely new in ALE 1.1:

- The Tag Memory API (Section 7)
- The Writing API (Section 9)
- The Logical Reader API (Section 10)
- The Access Control API (Section 11)

16.3 New Bindings

- A new HTTP over TLS (HTTPS) binding has been added for asynchronous notifications. See [ALE1.1Part2, Section 2.4].

16.4 Clarifications

- The state transitions in the lifecycle of an ECSpec have been clarified. See Section 5.6.
- The list of error conditions in Section 8.1.1 has been expanded to show that getStandardVersion and getVendorVersion each may raise an ImplementationException. In the ALE 1.0 specification, this was indicated in the SOAP binding but not in the main body of the specification.
- The UML descriptions of several parameters in the Reading API have been changed to match the XML binding.
- The description of the asynchronous notification mechanism has been formalized by introducing a formal "callback" interface at the UML level. The implementation at the binding level is exactly the same as in ALE 1.0.
- The treatment of names of ECSpecs and ECReports with respect to Unicode canonicalization rules has been clarified (in Section 4.5). It has also been clarified that the empty string may not be used as an ECSpec or ECReport name.
- The equivalence of null, omitted, and empty string values has been clarified, as has the equivalence of omitted and empty lists. See Section 4.7.
- The relationship of the result returned from getECSpec and the value originally provided to define has been clarified.

17 References

[ALE1.0] EPCglobal, "The Application Level Events (ALE) Specification, Version 1.0," EPCglobal Ratified Standard, September 2005, http://www.epcglobalinc.org/standards/ale/ale_1_0-standard-20050915.pdf.

[ALE1.1Part2] EPCglobal, "The Application Level Events (ALE) Specification, Version 1.1.1 Part II: XML and SOAP Bindings," EPCglobal Ratified Standard, March 2009, http://www.epcglobalinc.org/standards/ale/ale_1_1_1-standard-XMLandSOAPbindings-20090313.pdf.

[ASN.1] CCITT, "Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)", CCITT Recommendation X.209, January 1988.

[EPCAF] K. R. Traub et al, "EPCglobal Architecture Framework," EPCglobal technical document, July 2005, http://www.epcglobalinc.org/standards_technology/Final-epcglobal-arch-20050701.pdf.

[Gen2] EPCglobal, "EPC™ Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz – 960 MHz Version 1.1.0," EPCglobal Ratified Standard, October 2007, http://www.epcglobalinc.org/standards/uhfc1g2/uhfc1g2_1_1_0-standard-20071017.pdf.

[ISO15962] ISO/IEC, "Information technology – Radio frequency identification (RFID) for item management – Data protocol: data encoding rules and logical memory functions," ISO/IEC 15962:2004, October 2004.

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

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

[RFC3061] M. Mealling, "A URN Namespace of Object Identifiers," RFC3061, February 2001, http://www.ietf.org/rfc/rfc3061.

[TDS1.3.1] EPCglobal, "EPCglobal Tag Data Standards Version 1.3.1," EPCglobal Ratified Standard, September 2007, http://www.epcglobalinc.org/standards/tds/tds_1_3_1-standard-20070928.pdf.

[Unicode] The Unicode Consortium, *The Unicode Standard, Version 5.0*, Addison-Wesley, November, 2006, ISBN 0321480910.

18 Acknowledgement of Contributors and of Companies Opt'd-in during the Creation of this Standard (non-normative)

Disclaimer

Whilst every effort has been made to ensure that this document and the information contained herein are correct, EPCglobal and any other party involved in the creation of the document hereby state that the document is provided on an "as is" basis without *warranty, either expressed or implied, including but not limited to any warranty that the use of the information herein with not infringe any rights, of accuracy or fitness for purpose, and hereby disclaim any liability, direct or indirect, for damages or loss relating to the use of the document.*

Below is a list of active participants and contributors in the development of the ALE 1.1 specification. This list does not acknowledge those who only monitored the process without contributing or those who chose not to have their name listed here. An “active participant” for the purpose of this list is an individual who corresponded using the Working Group mailing list or who attended one or more face-to-face or teleconference meetings of the Working Group.

Mark Frey (EPCglobal Inc.), Facilitator

Richard Bach (GlobeRanger), Co-Chair, Conformance Requirements Editor

Daniel Paley (AWiD), Past Co-Chair

Bryan Tracey (GlobeRanger), Past Co-Chair

Ken Traub (Ken Traub Consulting LLC; BEA Systems), Co-Chair, Specification Editor

Soumya Roy Chowdhury (Polaris Networks), Test Plan Editor

Muhammad Alam (SAP Aktiengesellschaft)

Scott Barvick (Reva Systems)

Bud Biswas (Polaris Networks)

Daniel Bowman (Kimberly-Clark)

Rob Buck (Intermec)

Toby Cabot (Reva Systems)

Ching-Hsiang Chang (RFID Research Center, Chang Jung Christian University)

Rita Chatterjee (Cognizant Technology Solutions)

John Cooper (Kimberly-Clark)

Roberto DeVet (Target Corporation)

Paul Dietrich (Impinj)

Mustafa Dohadwala (Shipcom Wireless)

Reinhard Dunst (Elektrobit; 7iD Technologies)

Sastry Duri (IBM)

Suvojit Dutta (Cognizant Technology Solutions)

Nicholas Fergusson (EPCglobal Inc.)

Gerhard Gangl (Elektrobit; 7iD Technologies)

Greg Gilbert (Manhattan Associates)

Satyaki Gupta (Cognizant Technology Solutions)

Stephan Haller (SAP Aktiengesellschaft)

Craig Harmon (Q.E.D. Systems)

Sudhir Hasbe (SamSys)

Lars-Erik Helander (Intermec)

Jeremy Helm (ACSIS)

Yoshimura Hisato (Nippon Telegraph & Telephone Corp)

Marc Horowitz (BEA Systems)

Qiming Huang (Oracle Corporation)

Noriaki Itoh (Dai Nippon Printing)

Michael Jonas (Metro)

Howard Kapustein (Manhattan Associates)

Andreas Kerschbaumer (Elektrobit; 7iD Technologies)

Satoshi Kinoshita (NEC Corporation)

Yuval Kost (Sandlinks)

P. Krishna (Reva Systems)

Joe Kubler (Intermec)

Rakesh Kumar (Cognizant Technology Solutions)

Mi Young Kwak (Allixon Co., Ltd)

Ram Laks (rfXcel)

Mike Lange (Red Prairie)

Da-Gang Lee (Institute for Information Industry)

Kyungeun Lim (Research Center for Logistics Infoformation Technology)

Timo Liu (Regal Scan Tech)

Steve Lockhart (Sirit Technologies)

Jerome Louvel (Supply Insight)

Malena Mesarina (Hewlett-Packard)

Gena Morgan (EPCglobal Inc.)

Stephen Morris (Printronix)

David Nesbitt (Vue Technology)

Ted Osinski (MET Laboratories)

Cindy Osman (Sun Microsystems)

Giselle Ow-Yang (EPCglobal Inc.)

Sung Gong Park (MetaRights)

Jong Park (Tibco Software)

Nicolas Pauvre (GS1 France)

Eliot Polk (Reva Systems)

Elliot Polk (Reva Systems)

Jim Reed (MET Laboratories)

Kelly Rhoades (EPCglobal Inc.)

Teresa Rinella (Accenture)

John Ross (IBM)

Subhabrata Roy (Cognizant Technology Solutions)

Curt Rozeboom (Q.E.D. Systems)

Jeff Sailors (Intermec)

Avinava Sarkar (Cognizant Technology Solutions)

Rick Schendel (Target Corporation)

Chris Shabsin (BEA Systems)

Dave Shaw (Reva Systems)

Dong Cheul Shin (MetaRights)

Adam Sills (GlobeRanger)

Inderjeet Singh (Cognizant Technology Solutions)

Sylvia Stein (GS1 Netherlands)

Hiroki Tagato (NEC Corporation)

Wolfgang Thaller (Elektrobit; 7iD Technologies)

Phyllis Turner-Brim (Intermec)

Richard Ulrich (Wal-Mart Stores)

Nitin Vidwans (Wal-Mart Stores)

Margaret Wasserman (ThingMagic)

Steve Winkler (SAP Aktiengesellschaft)

Katsuyuki Yamashita (Nippon Telegraph & Telephone Corp)

James Zhang (TrueDemand Software)

The following list enumerates, in alphabetical order by company name, all companies that signed the EPCglobal IP Policy and the opt-in agreement for the EPCglobal Working Group that created the ALE 1.1 standard.

7iD Technologies (formerly EOSS GmbH)

Accenture
Acer Cybercenter Service Inc.
ACSIS
Afilias Limited
Allixon Co., Ltd
Altria Group, Inc./Kraft Foods
Alvin Systems
AMCO TEC International Inc.
AMOS Technologies Inc.
Applied Wireless (AWiD)
Auto-ID Labs - Cambridge
Auto-ID Labs - ICU
Auto-ID Labs - Japan
Auto-ID Labs - MIT
BEA Systems
Cheng-Loong Corporation
Cisco
City Univ of Hong Kong
Cognizant Technology Solutions
Convergence Sys Ltd
Dai Nippon Printing (DNP)
Denso Wave Inc
Elektrobit (formerly 7iD)
ECO, Inc.
EPCglobal Inc.
ETRI - Electronics and Telecommunication Research Institute
FEIG Electronics
France Telecom
Fujitsu Ltd
GlobeRanger
GS1 Australia EAN
GS1 Germany (CCG)

GS1 Hong Kong
GS1 International
GS1 Japan
GS1 Netherlands (EAN.nl)
GS1 South Korea
GS1 Sweden AB (EAN)
GS1 Taiwan (EAN)
GS1 UK
GS1 US
Hewlett-Packard Co. (HP)
IBM
Impinj
Institute for Information Industry
Intermec
Ken Traub Consulting LLC
Kimberly-Clark
KL-NET
KTNET - Korea Trade Network
Leiner Health Products Inc.
LG CNS
Research Center for Logistics Information Technology (LIT)
Lockheed Martin - Savi Technology Divison
Manhattan Associates
MET Laboratories
MetaBiz
MetaRights, Ltd.
Metro
Microelectronics Technology, Inc.
Mstar Semiconductor
NEC Corporation
Nippon Telegraph & Telephone Corp (NTT)
noFilis Ltd.

Nomura Research Institute
NXP Semiconductors
NYSSA S.R.L.
OatSystems
Oracle Corporation
Panda Logistics Co.Ltd
Pango Networks, Inc.
Polaris Networks
Polaris Networks
Printronix
Psion Teklogix Inc.
Q.E.D. Systems
Rafcore Systems Inc.
Red Prairie
Regal Scan Tech
RetailTech
Reva Systems
RF-IT Solutions GmbH
RFID Research Center, Chang Jung Christian University
rfXcel Corp
Samsung SDS
Sandlinks
SAP Aktiengesellschaft
Secure RF
Sedna Systems, Ltd.
Shipcom Wireless, Inc.
Sirit Technologies Inc
Sirit Technologies Inc
Supply Insight, Inc.
SupplyScape Corporation
Tagent Corporation
The Boeing Company

ThingMagic, LLC
Tibco Software, Inc
Toppan Printing Co., Ltd
Toray International, Inc.
Tracetracker Inovation AS
TrueDemand Software
Userstar Information System Co. Ltd
Ussen Limited Company
VeriSign
Vue Technology
Wal-Mart Stores, Inc.
Waldemar Winckel GmbH & Co. KG
Warelite Ltd