EPC Tag Data Standard

Version 1.5

Ratified on August 18th, 2010

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

The EPC Tag Data Standard defines the Electronic Product Code™, and also specifies the memory contents of Gen 2 RFID Tags. In more detail, the Tag Data Standard covers two broad areas:

- The specification of the Electronic Product Code, including its representation at various levels of the EPCglobal Architecture and it correspondence to GS1 keys and other existing codes.
- The specification of data that is carried on Gen 2 RFID tags, including the EPC, "user memory" data, control information, and tag manufacture information.

The Electronic Product Code is a universal identifier for any physical object. It is used in information systems that need to track or otherwise refer to physical objects. A very large subset of applications that use the Electronic Product Code also rely upon RFID Tags as a data carrier. For this reason, a large part of the Tag Data Standard is concerned with the encoding of Electronic Product Codes onto RFID tags, along with defining the standards for other data apart from the EPC that may be stored on a Gen 2 RFID tag.

Therefore, the two broad areas covered by the Tag Data Standard (the EPC and RFID) overlap in the parts where the encoding of the EPC onto RFID tags is discussed. Nevertheless, it should always be remembered that the EPC and RFID are not at all synonymous: EPC is an identifier, and RFID is a data carrier. RFID tags contain other data besides EPC identifiers (and in some applications may not carry an EPC identifier at all), and the EPC identifier exists in non-RFID contexts (those non-RFID contexts including the URI form used within information systems, printed human-readable EPC URIs, and EPC identifiers derived from bar code data following the procedures in this standard).

Audience for this document

The target audience for this specification includes:

- EPC Middleware vendors
- RFID Tag users and encoders
- Reader vendors
- Application developers
- System integrators

Differences From EPC Tag Data Standard Version 1.4

The EPC Tag Data Standard Version 1.5 is fully backward-compatible with EPC Tag Data Standard Version 1.4, with the exception of the definition of filter values as noted below.

The EPC Tag Data Standard Version 1.5 includes these new or enhanced features:

- The correspondence between certain special cases of GTIN and the SGTIN EPC have been clarified. This includes:
  - GTIN-12 and GTIN-13 (Section 7.1.1)
  - GTIN-8 and RCN-8 (Section 7.1.2)
  - Company Internal Numbering Trade Identification (GS1 Prefixes 04 and 0001 – 0007) (Section 7.1.3)
  - Restricted Circulation Trade Identification (GS1 Prefixes 02 and 20 – 29) (Section 7.1.4)
  - Coupon Code Identification for Restricted Distribution (GS1 Prefixes 05, 99, 981, and 982) (Section 7.1.5)
  - Refund Receipt (GS1 Prefix 980) (Section 7.1.6)
  - ISBN, ISMN, and ISSN (GS1 Prefixes 977, 978, or 979) (Section 7.1.7)
- The treatment of the '`0`' padding character in the GS1 Element String for the GRAI has been clarified (Section 7.4)
- Earlier versions of the Tag Data Standard incorrectly stated the upper limit on Location Extension in the SGLN-96 encoding procedure; this is now corrected.
- "Attribute Bits" have been introduced in the EPC Memory Bank of a Gen 2 RFID Tag. The Attribute Bits includes data that guides the handling of the object to which the tag is affixed, for example a bit that indicates the presence of hazardous material. (Section 11)
- The definitions of "Filter Values" for SGTIN and SSCC have been updated. In some instances, these changes are not backward-compatible with EPC Tag Data Standard Version 1.4. (Section 10)
- The EPC Tag URI and EPC Raw URI have been enhanced to include Attribute Bits, along with other control information introduced by the EPCglobal UHF Class 1 Gen 2 Air Interface SpecificationVersion 1.2.0. The latter includes the user memory indicator and the extended protocol control (XPC) bits. (Section 12)
- The contents of the TID Memory Bank of a Gen 2 RFID Tag are specified (Section 16)
- The framework for encoding of data elements into the User Memory Bank of a Gen 2 RFID Tag are specified. This framework is based on a new access method for ISO/IEC 15962 [ISO15962] called "Packed Objects," and will be included in the forthcoming 2nd edition of that specification. (Section 17)

In addition to the above new and enhanced features, the structure of the EPC Tag Data Standard has been completely revised. These revisions are intended to make the document clearer and more accessible to readers, as well as to better explain the conceptual underpinnings of the Tag Data Standard. Specifically, the revisions to the structure include the following:

- An up-front "roadmap" is included that shows how all the pieces of the Tag Data Standard fit together. (Section 3)
- The specification greatly expands on the topic of what is an Electronic Product Code, how is it used, how does it relate to the other EPC standards, and how does it relate to the GS1 keys defined in the GS1 General Specifications. (Section 4)
- Much more emphasis is laid on the Pure Identity EPC URI (Section 6), in the following ways:
  - The Pure Identity EPC URI is the basis for explaining of how the EPC is used.
  - The correspondence between GS1 Element Strings (as used in bar codes) and EPCs is described at the Pure Identity EPC URI level, not at the binary level. This allows this correspondence to be understood without reference to RFID Tags. (Section 7)
- The specification is clearly divided into those parts that are RFID-specific and those parts that are independent of RFID. In particular, it is emphasized that the EPC and RFID are not synonymous (an EPC may exist and be used in the absence of RFID, and an RFID tag may contain data other than an EPC).
- The description of the memory contents of a Gen 2 RFID Tag distinguishes between "control information" as distinct from "business data" and "tag manufacture information" (TID), and this concept is used throughout to help clarify how pieces fit together. In particular, this helps to describe role of Filter Values as data that is both distinct from the EPC and specific to the process of reading RFID Tags.
- The input to (output from) the encoding (decoding) procedures for the EPC Binary Encoding as used on RFID Tags is now expressed as an EPC Tag URI, rather than as a GS1 Element String.
- The encoding and decoding procedures for the EPC Binary Encoding are now more modular and table-driven, and less repetitive.

The changes above imply that the procedure to convert between a GS1 Element String and the EPC Binary Encoding as used on an RFID Tag is now described quite differently than in previous versions of the EPC Tag Data Standard. The net effect, however, is *identical* to the EPC Tag Data Standard Version 1.4 – no changes have been made to the encodings themselves, only their method of description.

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 `http://www.epcglobalinc.org/standards/` for more information.

This version of the Tag Data Standard is the fully Ratified version as Ratified by the EPCglobal Board ballot that was completed on August 18, 2010. Previously, this document had gone through all governance reviews and approvals of the previous version.

Comments on this document should be sent to the EPCglobal Software Action Group and addressed to **GS1help@gs1.org**.

Table of Contents

List of Figures

List of Tables

1 Introduction

The EPC Tag Data Standard defines the Electronic Product Code™, and also specifies the memory contents of Gen 2 RFID Tags. In more detail, the Tag Data Standard covers two broad areas:

- The specification of the Electronic Product Code, including its representation at various levels of the EPCglobal Architecture and it correspondence to GS1 keys and other existing codes.
- The specification of data that is carried on Gen 2 RFID tags, including the EPC, "user memory" data, control information, and tag manufacture information.

The Electronic Product Code is a universal identifier for any physical object. It is used in information systems that need to track or otherwise refer to physical objects. A very large subset of applications that use the Electronic Product Code also rely upon RFID Tags as a data carrier. For this reason, a large part of the Tag Data Standard is concerned with the encoding of Electronic Product Codes onto RFID tags, along with defining the standards for other data apart from the EPC that may be stored on a Gen 2 RFID tag.

Therefore, the two broad areas covered by the Tag Data Standard (the EPC and RFID) overlap in the parts where the encoding of the EPC onto RFID tags is discussed. Nevertheless, it should always be remembered that the EPC and RFID are not at all synonymous: EPC is an identifier, and RFID is a data carrier. RFID tags contain other data besides EPC identifiers (and in some applications may not carry an EPC identifier at all), and the EPC identifier exists in non-RFID contexts (those non-RFID contexts including the URI form used within information systems, printed human-readable EPC URIs, and EPC identifiers derived from bar code data following the procedures in this standard).

2 Terminology and Typographical Conventions

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

All sections of this document, with the exception of Section 1, 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 for illustrations of identifiers and other character strings that exist within information systems.

➢ 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.

The term “Gen 2 RFID Tag” (or just “Gen 2 Tag”) as used in this specification refers to any RFID tag that conforms to the EPCglobal UHF Class 1 Generation 2 Air Interface, Version 1.2.0 or later [UHFC1G2], as well as any RFID tag that conforms to another air interface standard that shares the same memory map. The latter includes specifications currently under development within EPCglobal such as the HF Class 1 Generation 2 Air Interface.

Bitwise addresses within Gen 2 Tag memory banks are indicated using hexadecimal numerals ending with a superscript “h”; for example, 20_h denotes bit address 20 hexadecimal (32 decimal).

3 Overview of Tag Data Standards

This section provides an overview of the Tag Data Standard and how the parts fit together.

The Tag Data Standard covers two broad areas:

- The specification of the Electronic Product Code, including its representation at various levels of the EPCglobal Architecture and it correspondence to GS1 keys and other existing codes.
- The specification of data that is carried on Gen 2 RFID tags, including the EPC, “user memory” data, control information, and tag manufacture information.

The Electronic Product Code is a universal identifier for any physical object. It is used in information systems that need to track or otherwise refer to physical objects. Within computer systems, including electronic documents, databases, and electronic messages, the EPC takes the form of an Internet Uniform Resource Identifier (URI). This is true regardless of whether the EPC was originally read from an RFID tag or some other kind of data carrier. This URI is called the “Pure Identity EPC URI.” The following is an example of a Pure Identity EPC URI:

```
urn:epc:id:sgtin:0614141.112345.400
```

A very large subset of applications that use the Electronic Product Code also rely upon RFID Tags as a data carrier. RFID is often a very appropriate data carrier technology to use for applications involving visibility of physical objects, because RFID permits data to be physically attached to an object such that reading the data is minimally invasive to material handling processes. For this reason, a large part of the Tag Data Standard is concerned with the encoding of Electronic Product Codes onto RFID tags, along with defining the standards for other data apart from the EPC that may be stored on a Gen 2 RFID tag. Owing to memory limitations of RFID tags, the EPC is not stored in URI form on the tag, but is instead encoded into a compact binary representation. This is called the “EPC Binary Encoding.”

Therefore, the two broad areas covered by the Tag Data Standard (the EPC and RFID) overlap in the parts where the encoding of the EPC onto RFID tags is discussed. Nevertheless, it should always be remembered that the EPC and RFID are not at all synonymous: EPC is an identifier, and RFID is a data carrier. RFID tags contain other data besides EPC identifiers (and in some applications may not carry an EPC identifier at all), and the EPC identifier exists in non-RFID contexts (those non-RFID contexts currently including the URI form used within information systems, printed human-readable EPC URIs, and EPC identifiers derived from bar code data following the procedures in this standard).

The term “Electronic Product Code” (or “EPC”) is used when referring to the EPC regardless of the concrete form used to represent it. The term “Pure Identity EPC URI” is used to refer specifically to the text form the EPC takes within computer systems, including electronic documents, databases, and electronic messages. The term “EPC Binary Encoding” is used specifically to refer to the form the EPC takes within the memory of RFID tags.

The following diagram illustrates the parts of the Tag Data Standard and how they fit together. (The colors in the diagram refer to the types of data that may be stored on RFID tags, explained further in Section 9.1.)

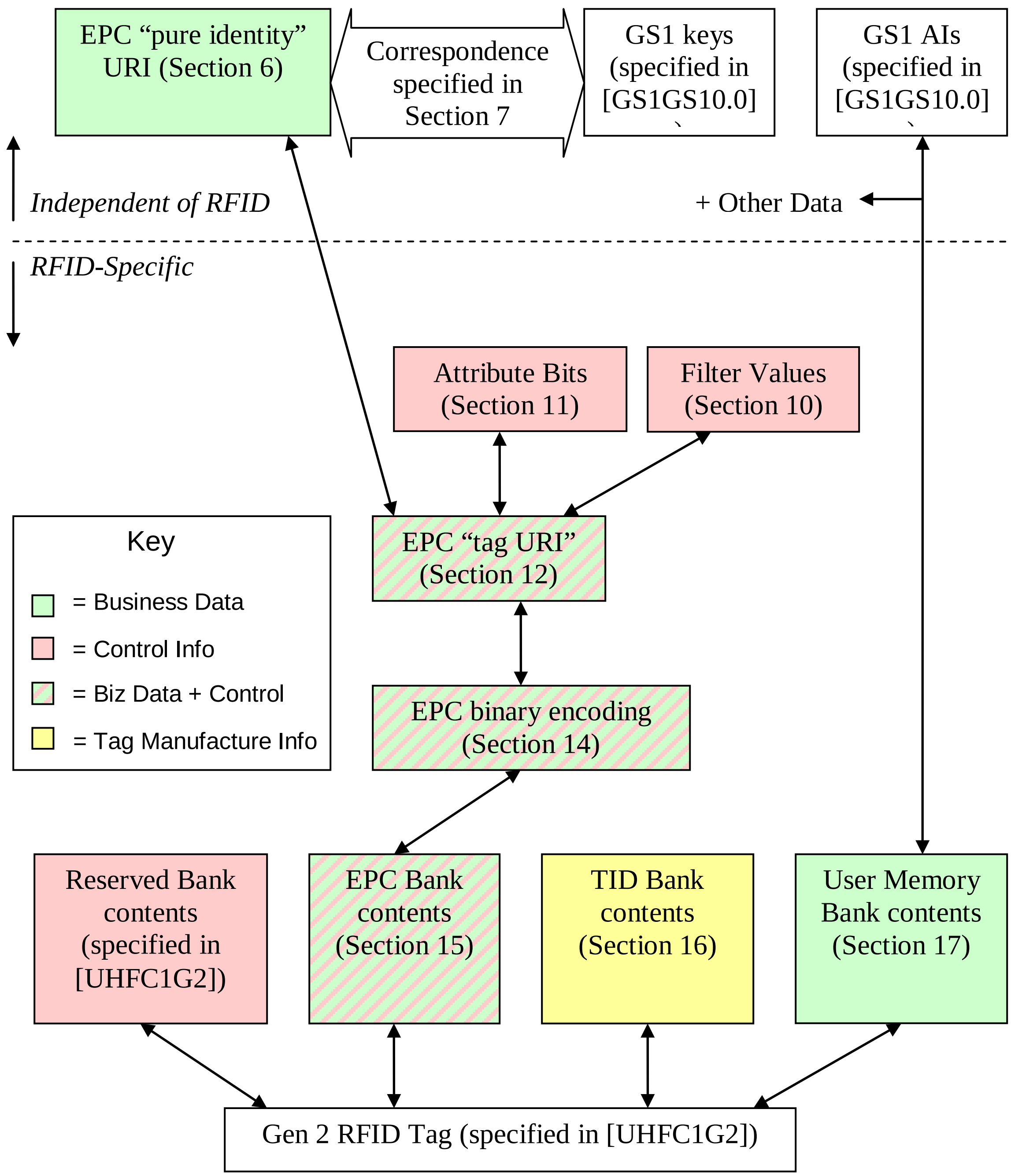


Figure 1. Organization of the EPC Tag Data Standard

The first few sections define those aspects of the Electronic Product Code that are independent from RFID.

Section 4 provides an overview of the Electronic Product Code (EPC) and how it relates to other EPCglobal standards and the GS1 General Specifications.

Section 6 specifies the Pure Identity EPC URI form of the EPC. This is a textual form of the EPC, and is recommended for use in business applications and business documents as a universal identifier for any physical object for which visibility information is kept. In particular, this form is what is used as the "what" dimension of visibility data in the EPC Information Services (EPCIS) specification, and is also available as an output from the Application Level Events (ALE) interface.

Section 7 specifies the correspondence between Pure Identity EPC URIs as defined in Section 6 and bar code element strings as defined in the GS1 General Specifications.

Section 8 specifies the Pure Identity Pattern URI, which is a syntax for representing sets of related EPCs, such as all EPCs for a given trade item regardless of serial number.

The remaining sections address topics that are specific to RFID, including RFID-specific forms of the EPC as well as other data apart from the EPC that may be stored on Gen 2 RFID tags.

Section 9 provides general information about the memory structure of Gen 2 RFID Tags.

Sections 10 and 11 specify "control" information that is stored in the EPC memory bank of Gen 2 tags along with a binary-encoded form of the EPC (EPC Binary Encoding). Control information is used by RFID data capture applications to guide the data capture process by providing hints about what kind of object the tag is affixed to. Control information is not part of the EPC, and does comprise any part of the unique identity of a tagged object. There are two kinds of control information specified: the "filter value" (Section 10) that makes it easier to read desired tags in an environment where there may be other tags present, such as reading a pallet tag in the presence of a large number of item-level tags, and "attribute bits" (Section 11) that provide additional special attribute information such as alerting to the presence of hazardous material. The same "attribute bits" are available regardless of what kind of EPC is used, whereas the available "filter values" are different depending on the type of EPC (and with certain types of EPCs, no filter value is available at all).

Section 12 specifies the "tag" Uniform Resource Identifiers, which is a compact string representation for the entire data content of the EPC memory bank of Gen 2 RFID Tags. This data content includes the EPC together with "control" information as defined in Sections 10 and 11. In the "tag" URI, the EPC content of the EPC memory bank is represented in a form similar to the Pure Identity EPC URI. Unlike the Pure Identity EPC URI, however, the "tag" URI also includes the control information content of the EPC memory bank. The "tag" URI form is recommended for use in capture applications that need to read control information in order to capture data correctly, or that need to write the full contents of the EPC memory bank. "Tag" URIs are used in the Application Level Events (ALE) interface, both as an input (when writing tags) and as an output (when reading tags).

Section 13 specifies the EPC Tag Pattern URI, which is a syntax for representing sets of related RFID tags based on their EPC content, such as all tags containing EPCs for a given range of serial numbers for a given trade item.

Sections 14 and 15 specify the contents of the EPC memory bank of a Gen 2 RFID tag at the bit level. Section 14 specifies how to translate between the the "tag" URI and the EPC Binary Encoding. The binary encoding is a bit-level representation of what is actually stored on the tag, and is also what is carried via the Low Level Reader Protocol

(LLRP) interface. Section 15 specifies how this binary encoding is combined with attribute bits and other control information in the EPC memory bank.

Section 16 specifies the binary encoding of the TID memory bank of Gen 2 RFID Tags.

Section 17 specifies the binary encoding of the User memory bank of Gen 2 RFID Tags.

4 The Electronic Product Code: A Universal Identifier for Physical Objects

The Electronic Product Code is designed to facilitiate business processes and applications that need to manipulate visibility data – data about observations of physical objects. The EPC is a universal identifier that provides a unique identity for any physical object. The EPC is designed to be unique across all physical objects in the world, over all time, and across all categories of physical objects. It is expressly intended for use by business applications that need to track all categories of physical objects, whatever they may be.

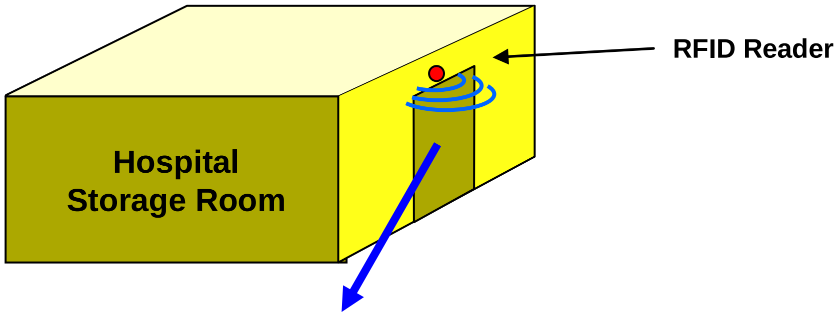
By contrast, seven GS1 identification keys defined in the GS1 General Specifications [GS1GS10.0] can identify categories of objects (GTIN), unique objects (SSCC, GLN, GIAI, GSRN), or a hybrid (GRAI, GDTI) that may identify either categories or unique objects depending on the absence or presence of a serial number. (Two other keys, GINC and GSIN, identify logical groupings, not physical objects.) The GTIN, as the only category identification key, requires a separate serial number to uniquely identify an object but that serial number is not considered part of the identification key.

There is a well-defined correspondence between EPCs and GS1 keys. This allows any physical object that is already identified by a GS1 key (or GS1 key + serial number combination) to be used in an EPC context where any category of physical object may be observed. Likewise, it allows EPC data captured in a broad visibility context to be correlated with other business data that is specific to the category of object involved and which uses GS1 keys.

The remainder of this section elaborates on these points.

4.1 The Need for a Universal Identifier: an Example

The following example illustrates how visibility data arises, and the role the EPC plays as a unique identifier for any physical object. In this example, there is a storage room in a hospital that holds radioactive samples, among other things. The hospital safety officer needs to track what things have been in the storage room and for how long, in order to ensure that exposure is kept within acceptable limits. Each physical object that might enter the storage room is given a unique Electronic Product Code, which is encoded onto an RFID Tag affixed to the object. An RFID reader positioned at the storage room door generates visibility data as objects enter and exit the room, as illustrated below.



Visibility Data Stream at Storage Room Entrance

Time	In / Out	EPC	Comment
8:23am	In	`urn:epc:id:sgtin:0614141.012345.62852`	10cc Syringe #62852 (trade item)
8:52am	In	`urn:epc:id:grai:0614141.54321.2528`	Pharma Tote #2528 (reusable transport)
8:59am	In	`urn:epc:id:sgtin:0614141.012345.1542`	10cc Syringe #1542 (trade item)
9:02am	Out	`urn:epc:id:giai:0614141.17320508`	Infusion Pump #52 (fixed asset)
9:32am	In	`urn:epc:id:gsrn:0614141.0000010253`	Nurse Jones (service relation)
9:42am	Out	`urn:epc:id:gsrn:0614141.0000010253`	Nurse Jones (service relation)
9:52am	In	`urn:epc:id:gdti:0614141.00001.1618034`	Patient Smith’s chart (document)

Figure 2. Example Visibility Data Stream

As the illustration shows, the data stream of interest to the safety officer is a series of events, each identifying a specific physical object and when it entered or exited the room. The unique EPC for each object is an identifier that may be used to drive the business process. In this example, the EPC (in Pure Identity EPC URI form) would be a primary key of a database that tracks the accumulated exposure for each physical object; each entry/exit event pair for a given object would be used to update the accumulated exposure database.

This example illustrates how the EPC is a single, *universal* identifier for any physical object. The items being tracked here include all kinds of things: trade items, reusable transports, fixed assets, service relations, documents, among others that might occur. By using the EPC, the application can use a single identifier to refer to any physical object, and it is not necessary to make a special case for each category of thing.

4.2 Use of Identifiers in a Business Data Context

Generally speaking, an identifier is a member of set (or “namespace”) of strings (names), such that each identifier is associated with a specific thing or concept in the real world. Identifiers are used within information systems to refer to the real world thing or concept in question. An identifier may occur in an electronic record or file, in a database, in an electronic message, or any other data context. In any given context, the producer and consumer must agree on which namespace of identifiers is to be used; within that context, any identifier belonging to that namespace may be used.

The keys defined in the GS1 General Specifications [GS1GS10.0] are each a namespace of identifiers for a particular category of real-world entity. For example, the Global Returnable Asset Identifier (GRAI) is a key that is used to identify returnable assets, such as plastic totes and pallet skids. The set of GRAI codes can be thought of as identifiers for the members of the set “all returnable assets.” A GRAI code may be used in a context where only returnable assets are expected; e.g., in a rental agreement from a moving services company that rents returnable plastic crates to customers to pack during a move. This is illustrated below.

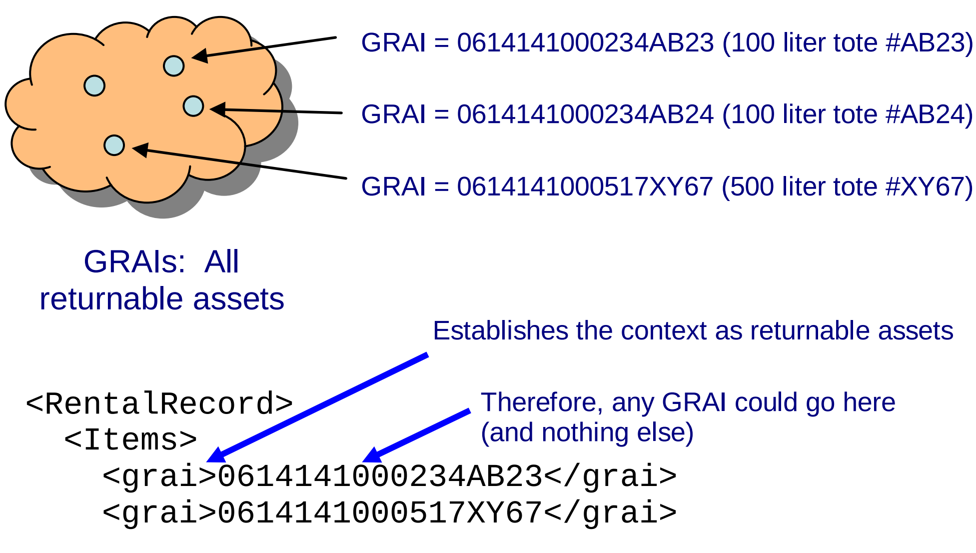
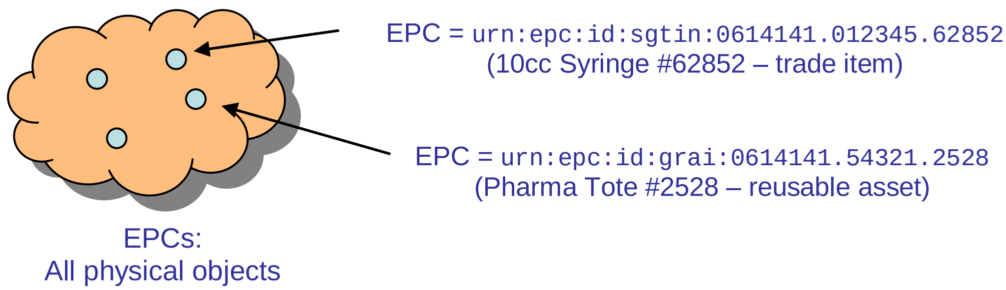


Figure 3. Illustration of GRAI Identifier Namespace

The upper part of the figure illustrates the GRAI identifier namespace. The lower part of the figure shows how a GRAI might be used in the context of a rental agreement, where only a GRAI is expected.



Establishes the context as all physical objects

Therefore, any EPC could go here

```
<EPCISDocument>
  <ObjectEvent>
    <epcList>

      <epc>urn:epc:id:sgtin:0614141.012345.62852</epc>
      <epc>urn:epc:id:grai:0614141.54321.2528</epc>
      …
```

Figure 4. Illustration of EPC Identifier Namespace

In contrast, the EPC namespace is a space of identifiers for *any* physical object. The set of EPCs can be thought of as identifiers for the members of the set “all physical objects.” EPCs are used in contexts where any type of physical object may appear, such as in the set of observations arising in the hospital storage room example above. Note that the EPC URI as illustrated in Figure 4 includes strings such as `sgtin`, `grai`, and so on as part of the EPC URI identifier. This is in contrast to GS1 Keys, where no such indication is part of the key itself (instead, this is indicated outside of the key, such as in the XML element name `<grai>` in the example in Figure 3, or in the Application Identifier (AI) that accompanies a GS1 Key in a GS1 Element String).

4.3 Relationship Between EPCs and GS1 Keys

There is a well-defined relationship between EPCs and GS1 keys. For each GS1 key that denotes an individual physical object (as opposed to a class), there is a corresponding EPC. This correspondence is formally defined by conversion rules specified in Section 7, which define how to map a GS1 key to the corresponding EPC value and vice versa. The well-defined correspondence between GS1 keys and EPCs allows for seamless migration of data between GS1 key and EPC contexts as necessary.

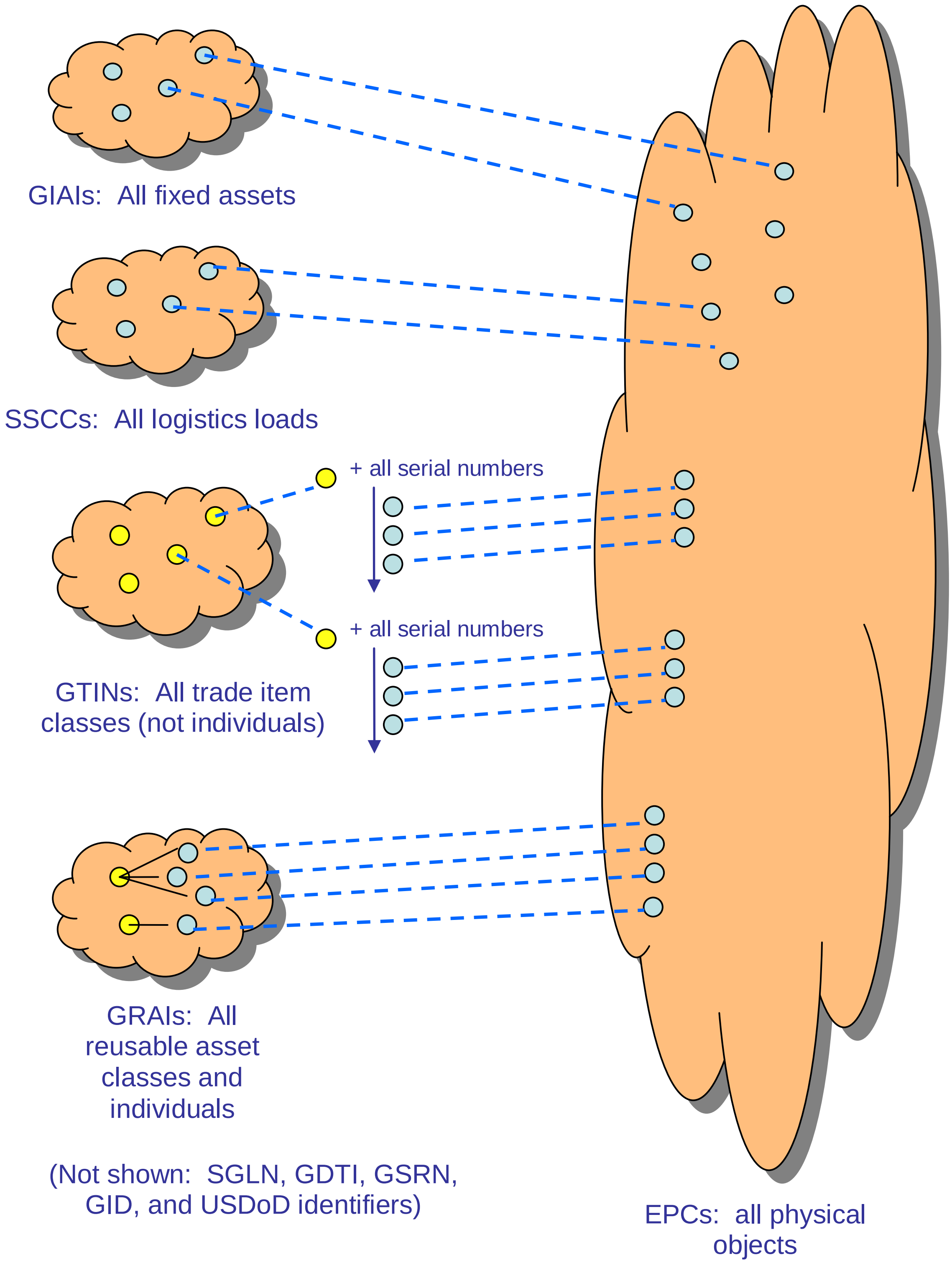


Figure 5. Illustration of Relationship of GS1 Key and EPC Identifier Namespaces

Not every GS1 key corresponds to an EPC, nor vice versa. Specifically:

- A Global Trade Identification Number (GTIN) by itself does not correspond to an EPC, because a GTIN identifies a *class* of trade items, not an individual trade item.

The combination of a GTIN and a unique serial number, however, *does* correspond to an EPC. This combination is called a Serialized Global Trade Identification Number, or SGTIN. The GS1 General Specifications do not define the SGTIN as a GS1 key.

- In the GS1 General Specifications, the Global Returnable Asset Identifier (GRAI) can be used to identify either a *class* of returnable assets, or an individual returnable asset, depending on whether the optional serial number is included. Only the form that includes a serial number, and thus identifies an individual, has a corresponding EPC. The same is true for the Global Document Type Identifier (GDTI).
- There is an EPC corresponding to each Global Location Number (GLN), and there is also an EPC corresponding to each combination of a GLN with an extension component. Collectively, these EPCs are referred to as Serialized Global Location Numbers (SGLNs).[1]
- EPCs include identifiers for which there is no corresponding GS1 key. These include the General Identifier and the US Department of Defense identifier.

The following table summarizes the EPC schemes defined in this specification and their correspondence to GS1 Keys.

EPC Scheme	Tag Encodings	Corresponding GS1 Key	Typical Use
`sgtin`	`sgtin-96` `sgtin-198`	GTIN key (plus added serial number)	Trade item
`sscc`	`sscc-96`	SSCC	Pallet load or other logistics unit load
`sgln`	`sgln-96` `sgln-195`	GLN key (with or without additional extension)	Location
`grai`	`grai-96` `grai-170`	GRAI (serial number mandatory)	Returnable/reusable asset
`giai`	`giai-96` `giai-202`	GIAI	Fixed asset
`gdti`	`gdti-96` `gdti-113`	GDTI (serial number mandatory)	Document
`gsrn`	`gsrn-96`	GSRN	Service relation (e.g., loyalty card)
`gid`	`gid-96`	[none]	Unspecified
`dod`	`dod-96`	[none]	US Dept of Defense supply chain

Table 1. EPC Schemes and Corresponding GS1 Keys

[1] The word "serialized" in this context is somewhat of a misnomer since a GLN without an extension also identifies a unique location, as opposed to a class of locations. The SGLN is intended to extend the capacity of the GLN. See [GS1GS10.0], Section 2.4.4, for limitations on use.

4.4 Use of the EPC in EPCglobal Architecture Framework

The EPCglobal Architecture Framework [EPCAF] is a collection of hardware, software, and data standards, together with shared network services that can be operated by EPCglobal, its delegates or third party providers in the marketplace, all in service of a common goal of enhancing business flows and computer applications through the use of Electronic Product Codes (EPCs). The EPCglobal Architecture Framework includes software standards at various levels of abstraction, from low-level interfaces to RFID reader devices all the way up to the business application level.

The EPC and related structures specified herein are intended for use at different levels within the EPCglobal architecture framework. Specifically:

- *Pure Identity EPC URI* The primary representation of an Electronic Product Code is as an Internet Uniform Resource Identifier (URI) called the Pure Identity EPC URI. The Pure Identity EPC URI is the preferred way to denote a specific physical object within business applications. The pure identity URI may also be used at the data capture level when the EPC is to be read from an RFID tag or other data carrier, in a situation where the additional "control" information present on an RFID tag is not needed.
- *EPC Tag URI* The EPC memory bank of a Gen 2 RFID Tag contains the EPC plus additional "control information" that is used to guide the process of data capture from RFID tags. The EPC Tag URI is a URI string that denotes a specific EPC together with specific settings for the control information found in the EPC memory bank. In other words, the EPC Tag URI is a text equivalent of the entire EPC memory bank contents. The EPC Tag URI is typically used at the data capture level when reading from an RFID tag in a situation where the control information is of interest to the capturing application. It is also used when writing the EPC memory bank of an RFID tag, in order to fully specify the contents to be written.
- *Binary Encoding* The EPC memory bank of a Gen 2 RFID Tag actually contains a compressed encoding of the EPC and additional "control information" in a compact binary form. There is a 1-to-1 translation between EPC Tag URIs and the binary contents of a Gen 2 RFID Tag. Normally, the binary encoding is only encountered at a very low level of software or hardware, and is translated to the EPC Tag URI or Pure Identity EPC URI form before being presented to application logic.

Note that the Pure Identity EPC URI is independent of RFID, while the EPC Tag URI and the Binary Encoding are specific to Gen 2 RFID Tags because they include RFID-specific "control information" in addition to the unique EPC identifier.

The figure below illustrates where these structures normally occur in relation to the layers of the EPCglobal Architecture Framework.

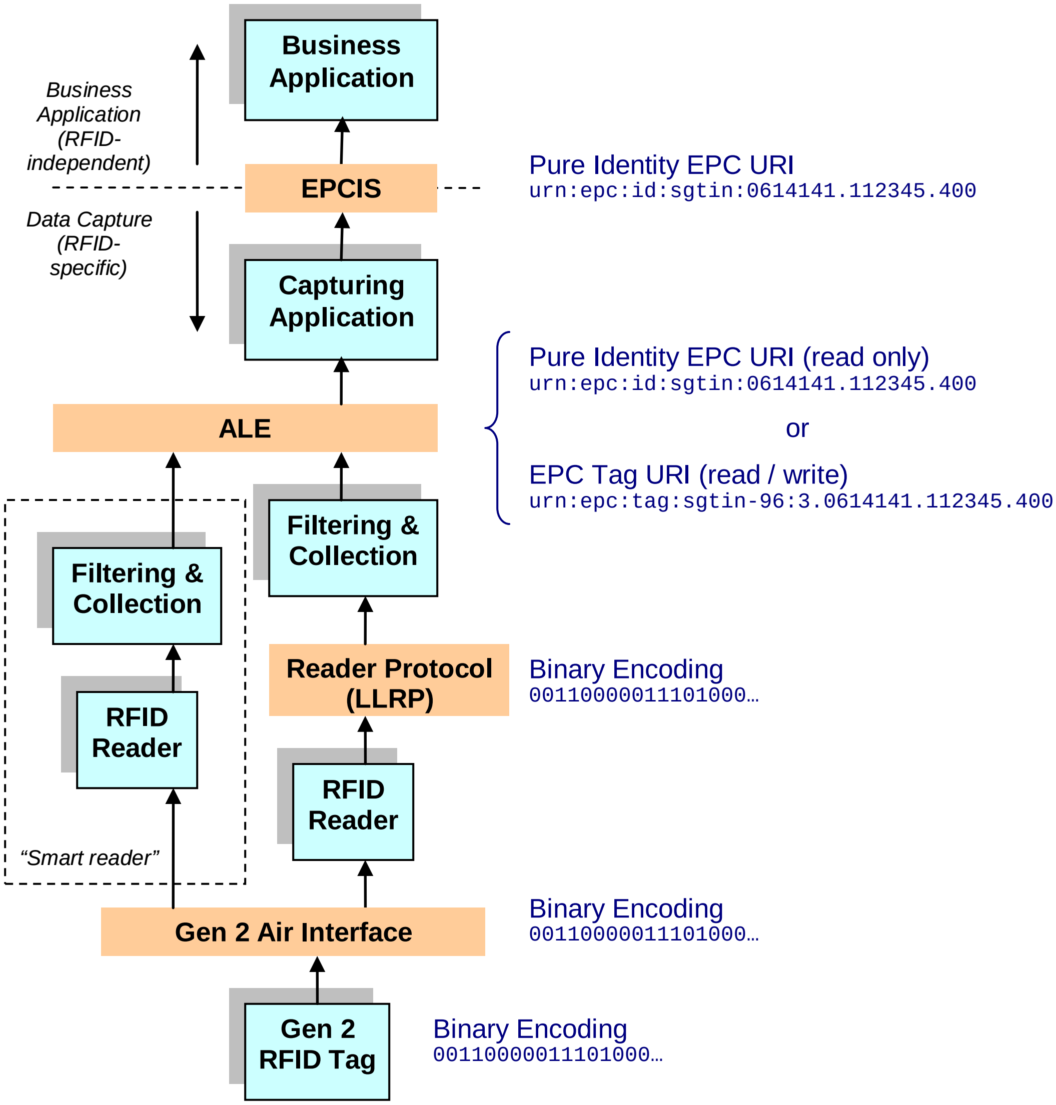


Figure 6. EPCglobal Architecture Framework and EPC Structures Used at Each Level

5 Common Grammar Elements

The syntax of various URI forms defined herein is specified via BNF grammars. The following grammar elements are used throughout this specification.

```
NumericComponent ::= ZeroComponent | NonZeroComponent
ZeroComponent ::= "0"
NonZeroComponent ::= NonZeroDigit Digit*
PaddedNumericComponent ::= Digit+
PaddedNumericComponentOrEmpty ::= Digit*
Digit ::= "0" | NonZeroDigit
```

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

UpperAlpha ::= “A” | “B” | “C” | “D” | “E” | “F” | “G”
             | “H” | “I” | “J” | “K” | “L” | “M” | “N”
             | “O” | “P” | “Q” | “R” | “S” | “T” | “U”
             | “V” | “W” | “X” | “Y” | “Z”

LowerAlpha ::= “a” | “b” | “c” | “d” | “e” | “f” | “g”
             | “h” | “i” | “j” | “k” | “l” | “m” | “n”
             | “o” | “p” | “q” | “r” | “s” | “t” | “u”
             | “v” | “w” | “x” | “y” | “z”

OtherChar ::= “!” | “’” | “(“ | “)“ | “*” | “+” | “,” | “-“
            | “.” | “:” | “;” | “=” | “_”

UpperHexChar ::= Digit | “A” | “B” | “C” | “D” | “E” | “F”

HexComponent ::= UpperHexChar+

Escape ::= “%” HexChar HexChar

HexChar ::= UpperHexChar | “a” | “b” | “c” | “d” | “e” |
“f”

GS3A3Char ::= Digit | UpperAlpha | LowerAlpha | OtherChar
            | Escape

GS3A3Component ::= GS3A3Char+
```

The syntactic construct `GS3A3Component` is used to represent fields of GS1 codes that permit alphanumeric and other characters as specified in Figure 3A3-1 of the GS1 General Specifications (see Appendix F). Owing to restrictions on URN syntax as defined by [RFC2141], not all characters permitted in the GS1 General Specifications may be represented directly in a URN. Specifically, the characters `"` (double quote), `%` (percent), `&` (ampersand), `/` (forward slash), `<` (less than), `>` (greater than), and `?` (question mark) are permitted in the GS1 General Specifications but may not be included directly in a URN. To represent one of these characters in a URN, escape notation must be used in which the character is represented by a percent sign, followed by two hexadecimal digits that give the ASCII character code for the character.

6 EPC URI

This section specifies the “pure identity URI” form of the EPC, or simply the “EPC URI.” The EPC URI is the preferred way within an information system to denote a specific physical object.

The EPC URI is a string having the following form:

`urn:epc:id:`*`scheme`*`:`*`component1`*`.`*`component2`*`.…`

where *`scheme`* names an EPC scheme, and *`component1`*, *`component2`*, and following parts are the remainder of the EPC whose precise form depends on which EPC scheme is used. The available EPC schemes are specified below in Table 2 in Section 6.3.

An example of a specific EPC URI is the following, where the scheme is `sgtin`:

`urn:epc:id:sgtin:0614141.112345.400`

Each EPC scheme provides a namespace of identifiers that can be used to identify physical objects of a particular type. Collectively, the EPC URIs from all schemes are unique identifiers for any type of physical object.

6.1 Use of the EPC URI

The EPC URI is the preferred way within an information system to denote a specific physical object.

The structure of the EPC URI guarantees worldwide uniqueness of the EPC across all types of physical objects and applications. In order to preserve worldwide uniqueness, each EPC URI must be used in its entirety when a unique identifier is called for, and not broken into constituent parts nor the `urn:epc:id:` prefix abbreviated or dropped.

When asking the question "do these two data structures refer to the same physical object?", where each data structure uses an EPC URI to refer to a physical object, the question may be answered simply by comparing the full EPC URI strings as specified in [RFC3986], Section 6.2. In most cases, the "simple string comparison" method sufficies, though if a URI contains percent-encoding triplets the hexadecimal digits may require case normalization as described in [RFC3986], Section 6.2.2.1. The construction of the EPC URI guarantees uniqueness across all categories of objects, provided that the URI is used in its entirety.

In other situations, applications may wish to exploit the internal structure of an EPC URI for purposes of filtering, selection, or distribution. For example, an application may wish to query a database for all records pertaining to instances of a specific product identified by a GTIN. This amounts to querying for all EPCs whose GS1 Company Prefix and item reference components match a given value, disregarding the serial number component. Another example is found in the Object Name Service (ONS) [ONS1.0.1], which uses the first component of an EPC to delegate a query to a "local ONS" operated by an individual company. This allows the ONS system to scale in a way that would be quite difficult if all ONS records were stored in a flat database maintained by a single organization.

While the internal structure of the EPC may be exploited for filtering, selection, and distribution as illustrated above, it is essential that the EPC URI be used in its entirety when used as a unique identifier.

6.2 Assignment of EPCs to Physical Objects

The act of allocating a new EPC and associating it with a specific physical object is called "commissioning." It is the responsibility of applications and business processes that commission EPCs to ensure that the same EPC is never assigned to two different physical objects; that is, to ensure that commissioned EPCs are unique. Typically, commissioning applications will make use of databases that record which EPCs have already been commissioned and which are still available. For example, in an application that commissions SGTINs by assigning serial numbers sequentially, such a database might record the last serial number used for each base GTIN.

Because visibility data and other business data that refers to EPCs may continue to exist long after a physical object ceases to exist, an EPC is ideally never reused to refer to a different physical object, even if the reuse takes place after the original object ceases to exist. There are certain situations, however, in which this is not possible; some of these are noted below. Therefore, applications that process historical data using EPCs should be prepared for the possibility that an EPC may be reused over time to refer to different physical objects, unless the application is known to operate in an environment where such reuse is prevented.

Seven of the EPC schemes specified herein correspond to GS1 keys, and so EPCs from those schemes are used to identify physical objects that have a corresponding GS1 key. When assigning these types of EPCs to physical objects, all relevant GS1 rules must be followed in addition to the rules specified herein. This includes the GS1 General Specifications [GS1GS10.0], the GTIN Allocation Rules, and so on. In particular, an EPC of this kind may only be commissioned by the licensee of the GS1 Company Prefix that is part of the EPC, or has been delegated the authority to do so by the GS1 Company Prefix licensee.

6.3 EPC URI Syntax

This section specifies the syntax of an EPC URI.

The formal grammar for the EPC URI is as follows:

```
EPC-URI ::= SGTIN-URI | SSCC-URI | SGLN-URI
          | GRAI-URI | GIAI-URI | GSRN-URI | GDTI-URI
          | GID-URI | EPCGID-URI | DOD-URI
```

where the various alternatives on the right hand side are specified in the sections that follow.

Each EPC URI scheme is specified in one of the following subsections, as follows:

EPC Scheme	Specified In	Corresponding GS1 Key	Typical Use
`sgtin`	Section 6.3.1	GTIN (with added serial number)	Trade item
`sscc`	Section 6.3.2	SSCC	Logistics unit
`sgln`	Section 6.3.3	GLN (with or without additional extension)	Location[2]

[2] While GLNs may be used to identify both locations and parties, the SGLN corresponds only to AI 414, which [GS1GS10.0] specifies is to be used to identify locations, and not parties.

EPC Scheme	Specified In	Corresponding GS1 Key	Typical Use
`grai`	Section 6.3.4	GRAI (serial number mandatory)	Returnable asset
`giai`	Section 6.3.5	GIAI	Fixed asset
`gdti`	Section 6.3.6	GDTI (serial number mandatory)	Document
`gsrn`	Section 6.3.7	GSRN	Service relation (e.g., loyalty card)
`gid`	Section 6.3.8	[none]	Unspecified
`usdod`	Section 6.3.9	[none]	US Dept of Defense supply chain

Table 2. EPC Schemes and Where the Pure Identity Form is Defined

6.3.1 Serialized Global Trade Item Number (SGTIN)

The Serialized Global Trade Item Number EPC scheme is used to assign a unique identity to an instance of a trade item, such as a specific instance of a product or SKU.

General syntax:

`urn:epc:id:sgtin:`*`CompanyPrefix`*`.`*`ItemReference`*`.`*`SerialNumber`*

Example:

```
urn:epc:id:sgtin:0614141.112345.400
```

Grammar:

```
SGTIN-URI ::= “urn:epc:id:sgtin:” SGTINURIBody
```

```
SGTINURIBody ::= 2*(PaddedNumericComponent “.”) GS3A3Component
```

The number of characters in the two `PaddedNumericComponent` fields must total 13 (not including any of the dot characters).

The Serial Number field of the SGTIN-URI is expressed as a `GS3A3Component`, which permits the representation of all characters permitted in the Application Identifier 21 Serial Number according to the GS1 General Specifications.[3] SGTIN-URIs that are derived from 96-bit tag encodings, however, will have Serial Numbers that consist only of digits and which have no leading zeros (unless the entire serial number consists of a single zero digit). These limitations are described in the encoding procedures, and in Section 12.3.1.

---

[3] As specified in Section 7.1, the serial number in the SGTIN is currently defined to be equivalent to AI 21 in the GS1 General Specifications. This equivalence is currently under discussion within GS1, and may be revised in future versions of the EPC Tag Data Standard.

The SGTIN consists of the following elements:

- The *GS1 Company Prefix*, assigned by GS1 to a managing entity or its delegates. This is the same as the GS1 Company Prefix digits within a GS1 GTIN key. See Section 7.1.2 for the case of a GTIN-8.
- The *Item Reference*, assigned by the managing entity to a particular object class. The Item Reference as it appears in the EPC URI is derived from the GTIN by concatenating the Indicator Digit of the GTIN (or a zero pad character, if the EPC URI is derived from a GTIN-8, GTIN-12, or GTIN-13) and the Item Reference digits, and treating the result as a single numeric string. See Section 7.1.2 for the case of a GTIN-8.
- The *Serial Number*, assigned by the managing entity to an individual object. The serial number is not part of the GTIN, but is formally a part of the SGTIN.

6.3.2 Serial Shipping Container Code (SSCC)

The Serial Shipping Container Code EPC scheme is used to assign a unique identity to a logistics handling unit, such as a the aggregate contents of a shipping container or a pallet load.

General syntax:

`urn:epc:id:sscc:`*`CompanyPrefix`*`.`*`SerialReference`*

Example:

`urn:epc:id:sscc:0614141.1234567890`

Grammar:

```
SSCC-URI ::= “urn:epc:id:sscc:” SSCCURIBody
```

```
SSCCURIBody ::= PaddedNumericComponent “.”
PaddedNumericComponent
```

The number of characters in the two `PaddedNumericComponent` fields must total 17 (not including any of the dot characters).

The SSCC consists of the following elements:

- The *GS1 Company Prefix*, assigned by GS1 to a managing entity. This is the same as the GS1 Company Prefix digits within a GS1 SSCC key.
- The *Serial Reference*, assigned by the managing entity to a particular logistics handling unit. The Serial Reference as it appears in the EPC URI is derived from the SSCC by concatenating the Extension Digit of the SSCC and the Serial Reference digits, and treating the result as a single numeric string.

6.3.3 Serialized Global Location Number (SGLN)

The Serialized Global Location Number EPC scheme is used to assign a unique identity to a physical location, such as a specific building or a specific unit of shelving within a warehouse.

General syntax:

`urn:epc:id:sgln:`*`CompanyPrefix`*`.`*`LocationReference`*`.`*`Extension`*

Example:

`urn:epc:id:sgln:0614141.12345.400`

Grammar:

```
SGLN-URI ::= “urn:epc:id:sgln:” SGLNURIBody
```

```
SGLNURIBody ::= PaddedNumericComponent “.”
PaddedNumericComponentOrEmpty “.” GS3A3Component
```

The number of characters in the two `PaddedNumericComponent` fields must total 12 (not including any of the dot characters).

The Extension field of the SGLN-URI is expressed as a `GS3A3Component`, which permits the representation of all characters permitted in the Application Identifier 254 Extension according to the GS1 General Specifications. SGLN-URIs that are derived from 96-bit tag encodings, however, will have Extensions that consist only of digits and which have no leading zeros (unless the entire extension consists of a single zero digit). These limitations are described in the encoding procedures, and in Section 12.3.1.

The SGLN consists of the following elements:

- The *GS1 Company Prefix*, assigned by GS1 to a managing entity. This is the same as the GS1 Company Prefix digits within a GS1 GLN key.
- The *Location Reference*, assigned uniquely by the managing entity to a specific physical location.
- The *GLN Extension*, assigned by the managing entity to an individual unique location. If the entire GLN Extension is just a single zero digit, it indicates that the SGLN stands for a GLN, without an extension.

6.3.4 Global Returnable Asset Identifier (GRAI)

The Global Returnable Asset Identifier EPC scheme is used to assign a unique identity to a specific returnable asset, such as a reusable shipping container or a pallet skid.

General syntax:

`urn:epc:id:grai:`*`CompanyPrefix`*`.`*`AssetType`*`.`*`SerialNumber`*

Example:

`urn:epc:id:grai:0614141.12345.400`

Grammar:

```
GRAI-URI ::= “urn:epc:id:grai:” GRAIURIBody
```

```
GRAIURIBody ::= PaddedNumericComponent “.”
PaddedNumericComponentOrEmpty “.” GS3A3Component
```


The number of characters in the two `PaddedNumericComponent` fields must total 12 (not including any of the dot characters).

The Serial Number field of the GRAI-URI is expressed as a `GS3A3Component`, which permits the representation of all characters permitted in the Serial Number according to the GS1 General Specifications. GRAI-URIs that are derived from 96-bit tag encodings, however, will have Serial Numbers that consist only of digits and which have no leading zeros (unless the entire serial number consists of a single zero digit). These limitations are described in the encoding procedures, and in Section 12.3.1.

The GRAI consists of the following elements:

- The *GS1 Company Prefix*, assigned by GS1 to a managing entity. This is the same as the GS1 Company Prefix digits within a GS1 GRAI key.
- The *Asset Type*, assigned by the managing entity to a particular class of asset.
- The *Serial Number*, assigned by the managing entity to an individual object. Because an EPC always refers to a specific physical object rather than an asset class, the serial number is mandatory in the GRAI-EPC.

6.3.5 Global Individual Asset Identifier (GIAI)

The Global Individual Asset Identifier EPC scheme is used to assign a unique identity to a specific asset, such as a forklift or a computer.

General syntax:

```
urn:epc:id:giai:CompanyPrefix.IndividulAssetReference
```

Example:

```
urn:epc:id:giai:0614141.12345400
```

Grammar:

```
GIAI-URI ::= “urn:epc:id:giai:” GIAIURIBody
GIAIURIBody ::= PaddedNumericComponent “.” GS3A3Component
```

The Individual Asset Reference field of the GIAI-URI is expressed as a `GS3A3Component`, which permits the representation of all characters permitted in the Serial Number according to the GS1 General Specifications. GIAI-URIs that are derived from 96-bit tag encodings, however, will have Serial Numbers that consist only of digits and which have no leading zeros (unless the entire serial number consists of a single zero digit). These limitations are described in the encoding procedures, and in Section 12.3.1.

The GIAI consists of the following elements:

- The *GS1 Company Prefix*, assigned by GS1 to a managing entity. The Company Prefix is the same as the GS1 Company Prefix digits within a GS1 GIAI key.
- The *Individual Asset Reference*, assigned uniquely by the managing entity to a specific asset.

6.3.6 Global Service Relation Number (GSRN)

The Global Service Relation Number EPC scheme is used to assign a unique identity to a service relation.

General syntax:

`urn:epc:id:gsrn:`*`CompanyPrefix`*`.`*`ServiceReference`*

Example:

`urn:epc:id:gsrn:0614141.1234567890`

Grammar:

```
GSRN-URI ::= “urn:epc:id:gsrn:” GSRNURIBody
```

```
GSRNURIBody ::= PaddedNumericComponent “.”
PaddedNumericComponent
```

The number of characters in the two `PaddedNumericComponent` fields must total 17 (not including any of the dot characters).

The GSRN consists of the following elements:

- The *GS1 Company Prefix*, assigned by GS1 to a managing entity. This is the same as the GS1 Company Prefix digits within a GS1 GSRN key.
- The *Service Reference*, assigned by the managing entity to a particular service relation.

6.3.7 Global Document Type Identifier (GDTI)

The Global Document Type Identifier EPC scheme is used to assign a unique identity to a specific document, such as land registration papers, an insurance policy, and others.

General syntax:

`urn:epc:id:gdti:`*`CompanyPrefix`*`.`*`DocumentType`*`.`*`SerialNumber`*

Example:

`urn:epc:id:gdti:0614141.12345.400`

Grammar:

```
GDTI-URI ::= “urn:epc:id:gdti:” GDTIURIBody
```

```
GDTIURIBody ::= PaddedNumericComponent “.”
PaddedNumericComponentOrEmpty “.” PaddedNumericComponent
```

The number of characters in the two `PaddedNumericComponent` fields must total 12 (not including any of the dot characters).

The Serial Number field of the GDTI-URI is expressed as a `NumericComponent`, which permits the representation of all characters permitted in the Serial Number according to the GS1 General Specifications. GDTI-URIs that are derived from 96-bit tag encodings, however, will have Serial Numbers that have no leading zeros (unless the entire serial number consists of a single zero digit). These limitations are described in the encoding procedures, and in Section 12.3.1.

The GDTI consists of the following elements:

- The *GS1 Company Prefix*, assigned by GS1 to a managing entity. This is the same as the GS1 Company Prefix digits within a GS1 GDTI key.
- The *Document Type*, assigned by the managing entity to a particular class of document.
- The *Serial Number*, assigned by the managing entity to an individual document. Because an EPC always refers to a specific document rather than a document class, the serial number is mandatory in the GDTI-EPC.

6.3.8 General Identifier (GID)

The General Identifier EPC scheme is independent of any specifications or identity scheme outside the EPCglobal Tag Data Standard.

General syntax:

`urn:epc:id:gid:`*`ManagerNumber`*`.`*`ObjectClass`*`.`*`SerialNumber`*

Example:

```
urn:epc:id:gid:95100000.12345.400
```

Grammar:

```
GID-URI ::= “urn:epc:id:gid:” GIDURIBody
GIDURIBody ::= 2*(NumericComponent “.”) NumericComponent
```

The GID consists of the following elements:

- The *General Manager Number* identifies an organizational entity (essentially a company, manager or other organization) that is responsible for maintaining the numbers in subsequent fields – Object Class and Serial Number. EPCglobal assigns the General Manager Number to an entity, and ensures that each General Manager Number is unique. Note that a General Manager Number is *not* a GS1 Company Prefix. A General Manager Number may only be used in GID EPCs.
- The *Object Class* is used by an EPC managing entity to identify a class or “type” of thing. These object class numbers, of course, must be unique within each General Manager Number domain.
- Finally, the *Serial Number* code, or serial number, is unique within each object class. In other words, the managing entity is responsible for assigning unique, non-repeating serial numbers for every instance within each object class.

6.3.9 US Department of Defense Identifier (DOD)

The US Department of Defense identifier is defined by the United States Department of Defense. This tag data construct may be used to encode 96-bit Class 1 tags for shipping goods to the United States Department of Defense by a supplier who has already been assigned a CAGE (Commercial and Government Entity) code.

At the time of this writing, the details of what information to encode into these fields is explained in a document titled "United States Department of Defense Supplier's Passive RFID Information Guide" that can be obtained at the United States Department of Defense's web site (http://www.dodrfid.org/supplierguide.htm).

Note that the DoD Guide explicitly recognizes the value of cross-branch, globally applicable standards, advising that “suppliers that are EPCglobal subscribers and possess a unique [GS1] Company Prefix may use any of the identity types and encoding instructions described in the EPC™ Tag Data Standards document to encode tags.”

General syntax:

`urn:epc:id:usdod:`*`CAGEOrDODAAC.SerialNumber`*

Example:

```
urn:epc:id:usdod:2S194.12345678901
```

Grammar:

```
DOD-URI ::= “urn:epc:id:usdod:” DODURIBody
DODURIBody ::= CAGECodeOrDODAAC “.” DoDSerialNumber
CAGECodeOrDODAAC ::= CAGECode | DODAAC
CAGECode ::= CAGECodeOrDODAACChar*5
DODAAC ::= CAGECodeOrDODAACChar*6
DoDSerialNumber ::= NumericComponent
CAGECodeOrDODAACChar ::= Digit | “A” | “B” | “C” | “D” |
“E” | “F” | “G” | “H” | “J” | “K” | “L” | “M” | “N” | “P” |
“Q” | “R” | “S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z”
```

7 Correspondence Between EPCs and GS1 Keys

As discussed in Section 4.3, there is a well-defined releationship between Electronic Product Codes (EPCs) and seven keys defined in the GS1 General Specifications [GS1GS10.0]. This section specifies the correspondence between EPCs and GS1 keys.

The correspondence between EPCs and GS1 keys relies on identifying the portion of a GS1 key that is the GS1 Company Prefix. The GS1 Company Prefix is a 6- to 11-digit number assigned by a GS1 Member Organization to a managing entity, and the managing entity is free to create GS1 keys using that GS1 Company Prefix.

In some instances, a GS1 Member Organization assigns a “one off” GS1 key, such as a complete GTIN, GLN, or other key, to a subscribing organization. In such cases, the GS1 Member Organization holds the GS1 Company Prefix, and therefore is responsible for identifying the number of digits that are to occupy the GS1 Company Prefix position within the EPC. The organization receiving the one-off key should consult with its GS1 Member Organization to determine the appropriate number of digits to ascribe to the

Company Prefix portion when constructing a corresponding EPC. In particular, a subscribing organization must *not* assume that the entire one-off key will occupy the Company Prefix digits of the EPC, unless specifically instructed by the GS1 Member Organization issuing the key. Moreover, a subscribing organization must *not* use the digits comprising a particular one-off key to construct any other kind of GS1 Key. For example, if a subscribing organization is issued a one-off GLN, it must *not* create SSCCs using the 12 digits of the one-off GLN as though it were a 12-digit GS1 Company Prefix.

When derived from GS1 Keys, the "first component of an EPC" is usually, but not always (e.g., GTIN-8, One-Off Key), a GS1 Company prefix. The GTIN-8 requires special treatment; see Section 7.1.2 for how an EPC is constructed from a GTIN-8. As stated above, the One-Off Key may or may not be used in its entirety as the first component of an EPC.

7.1 Serialized Global Trade Item Number (SGTIN)

The SGTIN EPC (Section 6.3.1) does not correspond directly to any GS1 key, but instead corresponds to a combination of a GTIN key plus a serial number. The serial number in the SGTIN is defined to be equivalent to AI 21 in the GS1 General Specifications.

The correspondence between the SGTIN EPC URI and a GS1 element string consisting of a GTIN key (AI 01) and a serial number (AI 21) is depicted graphically below:

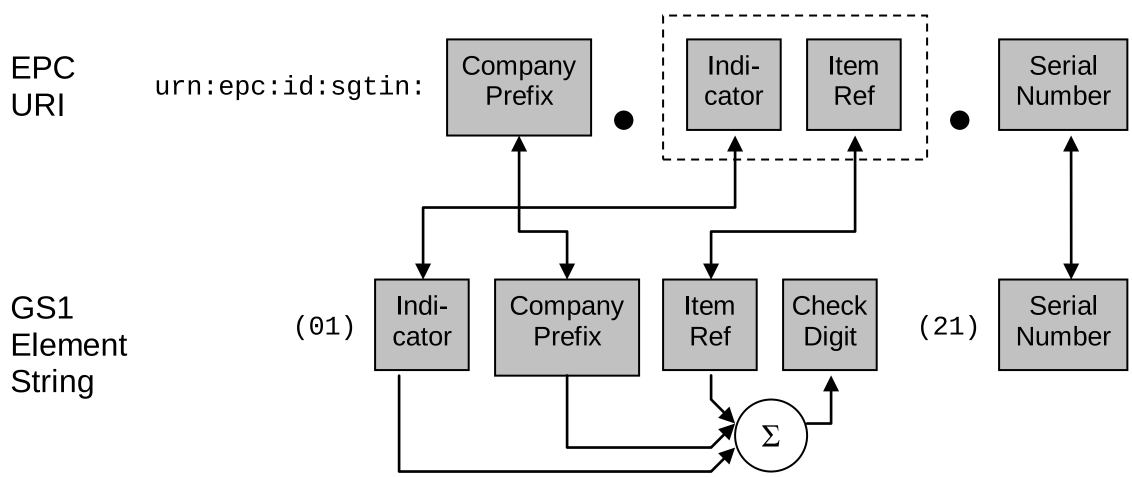


Figure 7. Correspondence between SGTIN EPC URI and GS1 Element String

(Note that in the case of a GTIN-12 or GTIN-13, a zero pad character takes the place of the Indicator Digit in the figure above.)

Formally, the correspondence is defined as follows. Let the EPC URI and the GS1 element string be written as follows:

EPC URI: $\texttt{urn:epc:id:sgtin:}d_2d_3\ldots d_{(L+1)}\texttt{.}d_1d_{(L+2)}d_{(L+3)}\ldots d_{13}\texttt{.}s_1s_2\ldots s_K$

GS1 Element String: $\texttt{(01)}d_1d_2\ldots d_{14}\ \texttt{(21)}s_1s_2\ldots s_K$

where $1 \le K \le 20$.

To find the GS1 element string corresponding to an SGTIN EPC URI:

1. Number the digits of the first two components of the EPC as shown above. Note that there will always be a total of 13 digits.
2. Number the characters of the serial number (third) component of the EPC as shown above. Each s_i corresponds to either a single character or to a percent-escape triplet consisting of a `%` character followed by two hexadecimal digit characters.
3. Calculate the check digit $d_{14} = (10 - ((3(d_1 + d_3 + d_5 + d_7 + d_9 + d_{11} + d_{13}) + (d_2 + d_4 + d_6 + d_8 + d_{10} + d_{12})) \bmod 10)) \bmod 10$.
4. Arrange the resulting digits and characters as shown for the GS1 Element String. If any s_i in the EPC URI is a percent-escape triplet `%xx`, in the GS1 Element String replace the triplet with the corresponding character according to Table 46 (Appendix A). (For a given percent-escape triplet `%xx`, find the row of Table 46 that contains `xx` in the "Hex Value" column; the "Graphic Symbol" column then gives the corresponding character to use in the GS1 Element String.)

To find the EPC URI corresponding to a GS1 element string that includes both a GTIN (AI 01) and a serial number (AI 21):

1. Number the digits and characters of the GS1 element string as shown above.
2. Except for a GTIN-8, determine the number of digits L in the GS1 Company Prefix. This may be done, for example, by reference to an external table of company prefixes. See Section 7.1.2 for the case of a GTIN-8.
3. Arrange the digits as shown for the EPC URI. Note that the GTIN check digit d_{14} is not included in the EPC URI. For each serial number character s_i, replace it with the corresponding value in the "URI Form" column of Table 46 (Appendix A) – either the character itself or a percent-escape triplet if s_i is not a legal URI character.

Example:

EPC URI: `urn:epc:id:sgtin:0614141.712345.32a%2Fb`

GS1 element string: `(01) 7 0614141 12345 1  (21) 32a/b`

Spaces have been added to the GS1 element string for clarity, but they are not normally present. In this example, the slash (`/`) character in the serial number must be represented as an escape triplet in the EPC URI.

7.1.1 GTIN-12 and GTIN-13

To find the EPC URI corresponding to the combination of a GTIN-12 or GTIN-13 and a serial number, first convert the GTIN-12 or GTIN-13 to a 14-digit number by adding two or one leading zero characters, respectively, as shown in [GS1GS10.0] Section 3.3.2.

Example:

GTIN-12: 614141 12345 2

Corresponding 14-digit number: 0 0614141 12345 2

Corresponding SGTIN-EPC: `urn:epc:id:sgtin:0614141.012345.`*`Serial`*

Example:

GTIN-13: 0614141 12345 2

Corresponding 14-digit number: 0 0614141 12345 2

Corresponding SGTIN-EPC: `urn:epc:id:sgtin:0614141.012345.`*`Serial`*

In these examples, spaces have been added to the GTIN strings for clarity, but are never encoded.

7.1.2 GTIN-8 and RCN-8

A GTIN-8 is a special case of the GTIN that is used to identify small trade items.

The GTIN-8 code consists of eight digits N_1, $N_2 \ldots N_8$, where the first digits N_1 to $N_L$are the GS1-8 Prefix (where L = 1, 2, or 3), the next digits N_{L+1} to N_7 are the Item Reference, and the last digit N_8 is the check digit. The GS1-8 Prefix is a one-, two-, or three-digit index number, administered by the GS1 Global Office. It does not identify the origin of the item. The Item Reference is assigned by the GS1 Member Organisation. The GS1 Member Organisations provide procedures for obtaining GTIN-8s.

To find the EPC URI corresponding to the combination of a GTIN-8 and a serial number, the following procedure SHALL be used. For the purpose of the procedure defined above in Section 7.1, the GS1 Company Prefix portion of the EPC shall be constructed by prepending five zeros to the first three digits of the GTIN-8; that is, the GS1 Company Prefix portion of the EPC is eight digits and shall be $00000N_1N_2N_3$. The Item Reference for the procedure shall be the remaining GTIN-8 digits apart from the check digit, that is, N_4 to N_7. The Indicator Digit for the procedure shall be zero.

Example:

GTIN-8: 95010939

Corresponding SGTIN-EPC: `urn:epc:id:sgtin:00000950.01093.`*`Serial`*

An RCN-8 is an 8-digit code beginning with GS1-8 Prefixes 0 or 2, as defined in [GS1GS10.0] Section 2.1.6.1. These are reserved for company internal numbering, and are not GTIN-8s. Such codes SHALL NOT be used to construct SGTIN EPCs, and the above procedure does not apply.

7.1.3 Company Internal Numbering (GS1 Prefixes 04 and 0001 – 0007)

The GS1 General Specifications reserve codes beginning with either 04 or 0001 through 0007 for company internal numbering. (See [GS1GS10.0], Sections 2.1.6.2 and 2.1.6.3.)

These numbers SHALL NOT be used to construct SGTIN EPCs. A future version of the EPCglobal Tag Data Standard may specify normative rules for using Company Internal Numbering codes in EPCs.

7.1.4 Restricted Circulation (GS1 Prefixes 02 and 20 – 29)

The GS1 General Specifications reserve codes beginning with either 02 or 20 through 29 for restricted circulation for geopolitical areas defined by GS1 member organizations and for variable measure trade items. (See [GS1GS10.0], Sections 2.1.6.4 and 2.1.7.)

These numbers SHALL NOT be used to construct SGTIN EPCs. A future version of the EPCglobal Tag Data Standard may specify normative rules for using Restricted Circulation codes in EPCs.

7.1.5 Coupon Code Identification for Restricted Distribution (GS1 Prefixes 05, 99, 981, and 982)

Coupons may be identified by constructing codes according to Sections 2.6.3, 2.6.4, and 2.6.5 of the GS1 General Specifications. The resulting numbers begin with GS1 Prefixes 05, 99, 981, or 982. Strictly speaking, however, a coupon is not a trade item, and these coupon codes are not actually trade item identification numbers.

Therefore, coupon codes SHALL NOT be used to construct SGTIN EPCs.

7.1.6 Refund Receipt (GS1 Prefix 980)

Section 2.6.8 of the GS1 General Specification specifies the construction of codes to represent refund receipts, such as those created by bottle recycling machines for redemption at point-of-sale. The resulting number begins with GS1 Prefix 980. Strictly speaking, however, a refund receipt is not a trade item, and these refund receipt codes are not actually trade item identification numbers.

Therefore, refund receipt codes SHALL NOT be used to construct SGTIN EPCs.

7.1.7 ISBN, ISMN, and ISSN (GS1 Prefixes 977, 978, or 979)

The GS1 General Specifications provide for the use of a 13-digit identifier to represent International Standard Book Number, International Standard Music Number, and International Standard Serial Number codes. The resulting code looks like a GTIN whose GS1 Prefix is 977, 978, or 979.

A study group has been established by GS1 with the book industry. That group will determine end user requirements for the usage of ISBN, ISMN and ISSN in EPCs. Therefore, such numbers SHALL NOT be used to construct SGTIN EPCs at this time. A future version of the EPCglobal Tag Data Standard will specify normative rules for using ISBN, ISMN and ISSN codes in SGTIN EPCs.

Explanation (non-normative): ISBN, ISMN, and ISSN codes are used for books, printed music, and periodical publications, respectively. The codes are defined by ISO and administered by the International ISBN Agency and affiliated national registration agencies. ISSN is a separate organization (http://www.issn.org/) and ISMN also (http://www.ismn-international.org/). While ISBN and ISMN codes are assigned outside the GS1 System, they may be represented as GTINs by prefixing the ISBN or ISMN code with 978 or 979. Because they are assigned outside the GS1 System it is not clear how to apply the SGTIN EPC encoding rules.

While these codes are not assigned by GS1, they have a very similar internal structure that readily lends itself to similar treatment when creating EPCs. An ISBN code consists of the following parts, shown below with the corresponding concept from the GS1 system:

Registrant Group Element	=	*GS1 Prefix (978 or 979 plus more digits)*
Registrant Element	=	*Remainder of GS1 Company Prefix*
Publication Element	=	*Item Reference*
Check Digit	=	*Check Digit*

The Registrant Group Elements are assigned to ISBN registration agencies, who in turn assign Registrant Elements to publishers, who in turn assign Publication Elements to individual publication editions. This exactly parallels the construction of GTIN codes. As in GTIN, the various components are of variable length, and as in GTIN, each publisher knows the combined length of the Registrant Group Element and Registrant Element, as the combination is assigned to the publisher. Happily, the total length of the "978" or "979" prefix, the Registrant Group Element, and the Registrant Element is in the range of 6 to 12 digits, which is exactly the range of company prefix lengths permitted in the SGTIN EPC. This suggests a natural way of handing ISBN codes. In The Netherlands there is now a pilot were they use partition value '0' to handle this. There are also some other rules for handling ISBN's. For example an ISBN stays with the combination Author Title, even when the Author changes Publisher.

A study group has been established by GS1 with the book industry. That group will determine end user requirements for the usage of ISBN, ISMN and ISSN in tags. The result may be to adopt a scheme as suggested by the above considerations.

7.2 Serial Shipping Container Code (SSCC)

The SSCC EPC (Section 6.3.2) corresponds directly to the SSCC key defined in Sections 2.2.1 and 3.3.1 of the GS1 General Specifications [GS1GS10.0].

The correspondence between the SSCC EPC URI and a GS1 element string consisting of an SSCC key (AI 00) is depicted graphically below:

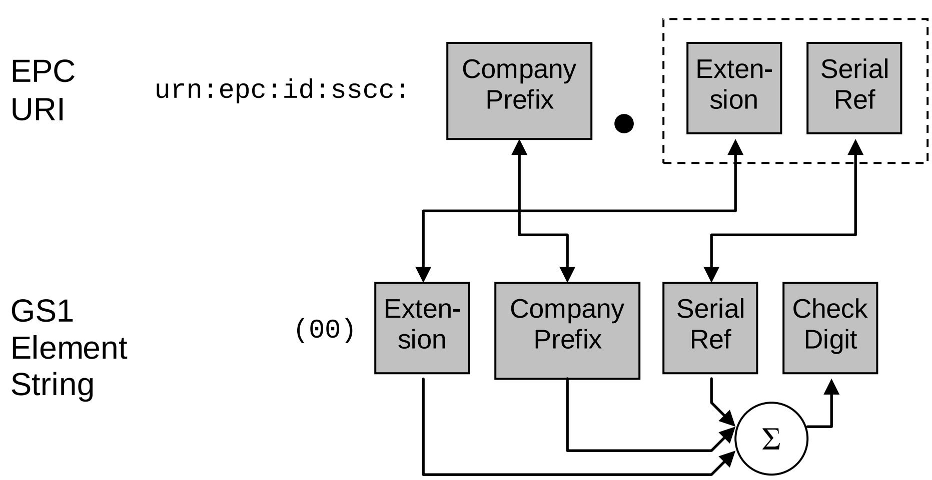


Figure 8. Correspondence between SSCC EPC URI and GS1 Element String

Formally, the correspondence is defined as follows. Let the EPC URI and the GS1 element string be written as follows:

EPC URI: `urn:epc:id:sscc:`$d_2 d_3 \ldots d_{(L+1)}.d_1 d_{(L+2)} d_{(L+3)} \ldots d_{17}$

GS1 Element String: `(00)`$d_1 d_2 \ldots d_{18}$

To find the GS1 element string corresponding to an SSCC EPC URI:

1. Number the digits of the two components of the EPC as shown above. Note that there will always be a total of 17 digits.
2. Calculate the check digit $d_{18} = (10 - ((3(d_1 + d_3 + d_5 + d_7 + d_9 + d_{11} + d_{13} + d_{15} + d_{17}) + (d_2 + d_4 + d_6 + d_8 + d_{10} + d_{12} + d_{14} + d_{16})) \bmod 10)) \bmod 10$.
3. Arrange the resulting digits and characters as shown for the GS1 Element String.

To find the EPC URI corresponding to a GS1 element string that includes an SSCC (AI 00):

1. Number the digits and characters of the GS1 element string as shown above.
2. Determine the number of digits *L* in the GS1 Company Prefix. This may be done, for example, by reference to an external table of company prefixes.
3. Arrange the digits as shown for the EPC URI. Note that the SSCC check digit d_{18} is not included in the EPC URI.

Example:

EPC URI: `urn:epc:id:sscc:0614141.1234567890`

GS1 element string: `(00) 1 0614141 234567890 8`

Spaces have been added to the GS1 element string for clarity, but they are never encoded.

7.3 Serialized Global Location Number (SGLN)

The SGLN EPC (Section 6.3.3) corresponds either directly to a Global Location Number key (GLN) as specified in Sections 2.4.4 and 3.7.9 of the GS1 General Specifications [GS1GS10.0], or to the combination of a GLN key plus an extension number as specified in Section 3.5.10 of [GS1GS10.0]. An extension number of zero is reserved to indicate that an SGLN EPC denotes an unextended GLN, rather than a GLN plus extension.

The correspondence between the SGLN EPC URI and a GS1 element string consisting of a GLN key (AI 414) *without* an extension is depicted graphically below:

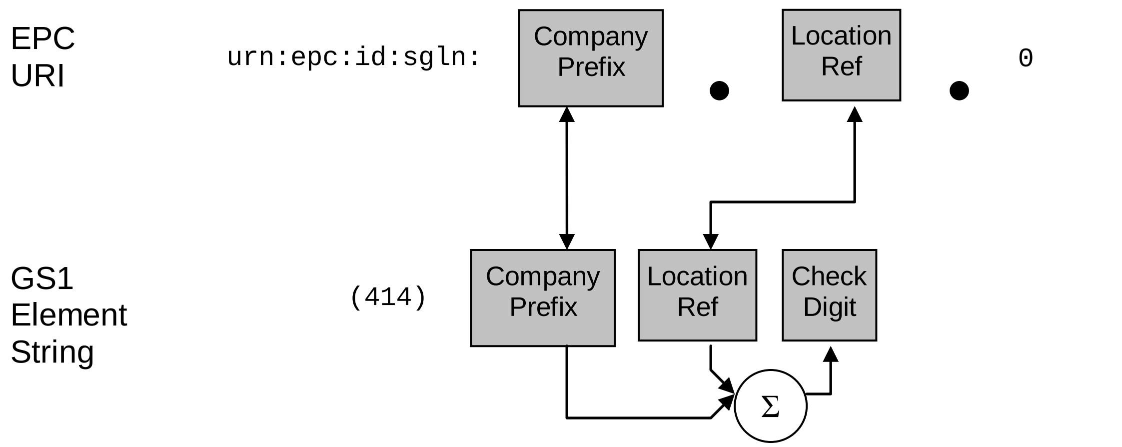


Figure 9. Correspondence between SGLN EPC URI without extension and GS1 Element String

The correspondence between the SGLN EPC URI and a GS1 element string consisting of a GLN key (AI 414) together with an extension (AI 254) is depicted graphically below:

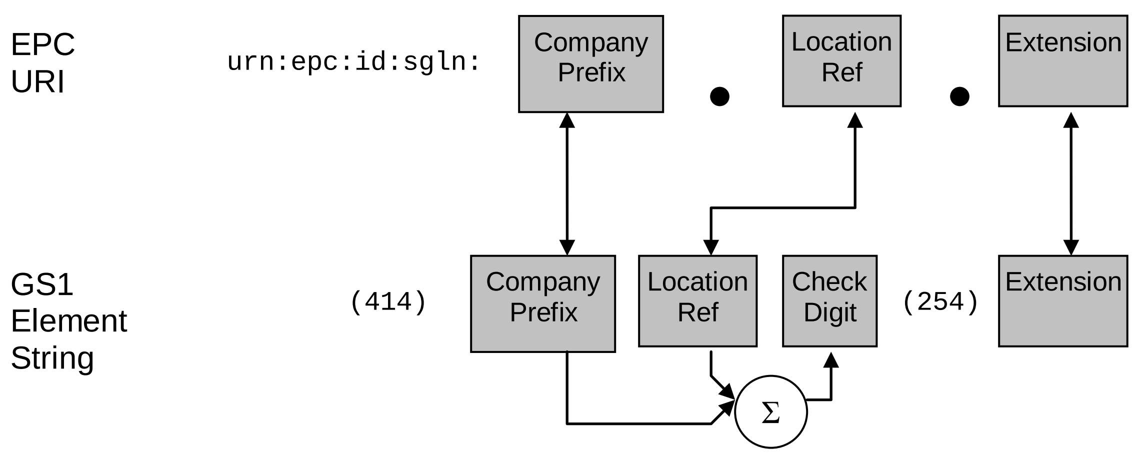


Figure 10. Correspondence between SGLN EPC URI with extension and GS1 Element String

Formally, the correspondence is defined as follows. Let the EPC URI and the GS1 element string be written as follows:

EPC URI: `urn:epc:id:sgln:`$d_1 d_2 \ldots d_L . d_{(L+1)} d_{(L+2)} \ldots d_{12} . s_1 s_2 \ldots s_K$

GS1 Element String: `(414)`$d_1 d_2 \ldots d_{13}$ `(254)`$s_1 s_2 \ldots s_K$

To find the GS1 element string corresponding to an SGLN EPC URI:

1. Number the digits of the first two components of the EPC as shown above. Note that there will always be a total of 12 digits.
2. Number the characters of the serial number (third) component of the EPC as shown above. Each s_i corresponds to either a single character or to a percent-escape triplet consisting of a `%` character followed by two hexadecimal digit characters.

3. Calculate the check digit $d_{13} = (10 - ((3(d_2 + d_4 + d_6 + d_8 + d_{10} + d_{12}) + (d_1 + d_3 + d_5 + d_7 + d_9 + d_{11})) \bmod 10)) \bmod 10$.

4. Arrange the resulting digits and characters as shown for the GS1 Element String. If any s_i in the EPC URI is a percent-escape triplet `%xx`, in the GS1 Element String replace the triplet with the corresponding character according to Table 46 (Appendix A). (For a given percent-escape triplet `%xx`, find the row of Table 46 that contains `xx` in the "Hex Value" column; the "Graphic Symbol" column then gives the corresponding character to use in the GS1 Element String.). If the serial number consists of a single character s_1 and that character is the digit zero ('`0`'), omit the extension from the GS1 Element String.

To find the EPC URI corresponding to a GS1 element string that includes a GLN (AI 414), with or without an accompanying extension (AI 254):

1. Number the digits and characters of the GS1 element string as shown above.

2. Determine the number of digits *L* in the GS1 Company Prefix. This may be done, for example, by reference to an external table of company prefixes.

3. Arrange the digits as shown for the EPC URI. Note that the GLN check digit d_{13} is not included in the EPC URI. For each serial number character s_i, replace it with the corresponding value in the "URI Form" column of Table 46 (Appendix A) – either the character itself or a percent-escape triplet if s_i is not a legal URI character. If the input GS1 element string did not include an extension (AI 254), use a single zero digit ('`0`') as the entire serial number $s_1 s_2 \ldots s_K$ in the EPC URI.

Example (without extension):

EPC URI: `urn:epc:id:sgln:0614141.12345.0`

GS1 element string: `(414) 0614141 12345 2`

Example (with extension):

EPC URI: `urn:epc:id:sgln:0614141.12345.32a%2Fb`

GS1 element string: `(414) 7 0614141 12345 2  (254) 32a/b`

Spaces have been added to the GS1 element string for clarity, but they are never encoded. In this example, the slash (`/`) character in the serial number must be represented as an escape triplet in the EPC URI.

7.4 Global Returnable Asset Identifier (GRAI)

The GRAI EPC (Section 6.3.4) corresponds directly to a serialized GRAI key defined in Sections 2.3.1 and 3.9.3 of the GS1 General Specifications [GS1GS10.0]. Because an EPC always identifies a specific physical object, only GRAI keys that include the optional serial number have a corresponding GRAI EPC. GRAI keys that lack a serial number refer to asset classes rather than specific assets, and therefore do not have a corresponding EPC (just as a GTIN key without a serial number does not have a corresponding EPC).

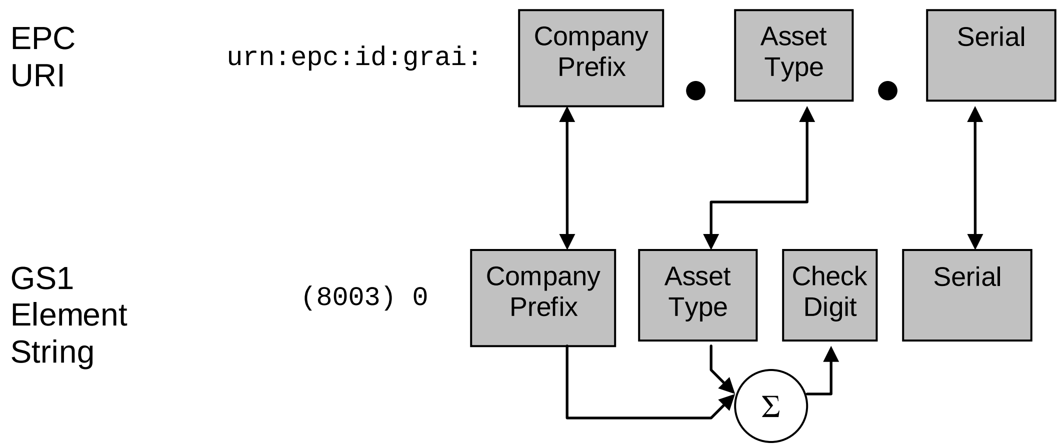


Figure 11.Correspondence between GRAI EPC URI and GS1 Element String

Note that the GS1 Element String includes an extra zero (‘`0`’) digit following the Application Identifier `(8003)`. This zero digit is extra padding in the element string, and is *not* part of the GRAI key itself.

Formally, the correspondence is defined as follows. Let the EPC URI and the GS1 element string be written as follows:

EPC URI: `urn:epc:id:grai:`$d_1 d_2 \ldots d_L . d_{(L+1)} d_{(L+2)} \ldots d_{12} . s_1 s_2 \ldots s_K$

GS1 Element String: `(8003)0`$d_1 d_2 \ldots d_{13} s_1 s_2 \ldots s_K$

To find the GS1 element string corresponding to a GRAI EPC URI:

1. Number the digits of the first two components of the EPC as shown above. Note that there will always be a total of 12 digits.
2. Number the characters of the serial number (third) component of the EPC as shown above. Each s_i corresponds to either a single character or to a percent-escape triplet consisting of a `%` character followed by two hexadecimal digit characters.
3. Calculate the check digit $d_{13} = (10 - ((3(d_2 + d_4 + d_6 + d_8 + d_{10} + d_{12}) + (d_1 + d_3 + d_5 + d_7 + d_9 + d_{11})) \bmod 10)) \bmod 10$.
4. Arrange the resulting digits and characters as shown for the GS1 Element String. If any s_i in the EPC URI is a percent-escape triplet `%xx`, in the GS1 Element String replace the triplet with the corresponding character according to Table 46 (Appendix A). (For a given percent-escape triplet `%xx`, find the row of Table 46 that contains `xx` in the “Hex Value” column; the “Graphic Symbol” column then gives the corresponding character to use in the GS1 Element String.).

To find the EPC URI corresponding to a GS1 element string that includes a GRAI (AI 8003):

1. If the number of characters following the `(8003)` application identifier is less than or equal to 14, stop: this element string does not have a corresponding EPC because it does not include the optional serial number.

2. Number the digits and characters of the GS1 element string as shown above.
3. Determine the number of digits *L* in the GS1 Company Prefix. This may be done, for example, by reference to an external table of company prefixes.
4. Arrange the digits as shown for the EPC URI. Note that the GRAI check digit d_{13} is not included in the EPC URI. For each serial number character s_i, replace it with the corresponding value in the "URI Form" column of Table 46 (Appendix A) – either the character itself or a percent-escape triplet if s_i is not a legal URI character.

Example:

EPC URI: `urn:epc:id:grai:0614141.12345.32a%2Fb`

GS1 element string: `(8003) 0 0614141 12345 2 32a/b`

Spaces have been added to the GS1 element string for clarity, but they are never encoded. In this example, the slash (`/`) character in the serial number must be represented as an escape triplet in the EPC URI.

7.5 Global Individual Asset Identifier (GIAI)

The GIAI EPC (Section 6.3.5) corresponds directly to the GIAI key defined in Sections 2.3.2 and 3.9.4 of the GS1 General Specifications [GS1GS10.0].

The correspondence between the GIAI EPC URI and a GS1 element string consisting of a GIAI key (AI 8004) is depicted graphically below:

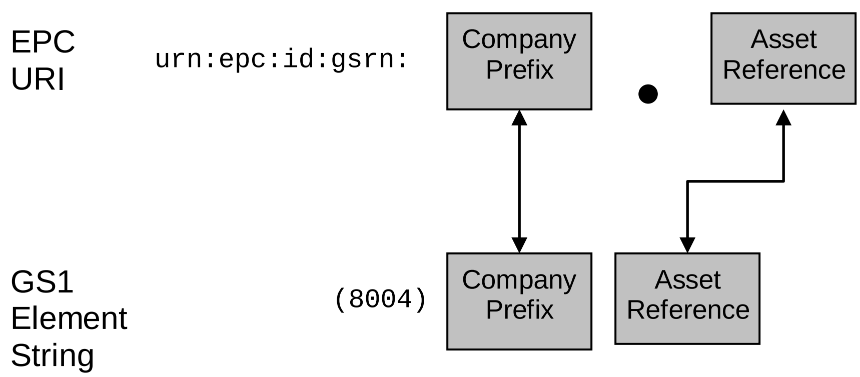


Figure 12.Correspondence between GIAI EPC URI and GS1 Element String

Formally, the correspondence is defined as follows. Let the EPC URI and the GS1 element string be written as follows:

EPC URI: `urn:epc:id:giai:`$d_1d_2\ldots d_L.s_1s_2\ldots s_K$

GS1 Element String: `(8004)`$d_1d_2\ldots d_Ls_1s_2\ldots s_K$

To find the GS1 element string corresponding to a GIAI EPC URI:

1. Number the characters of the two components of the EPC as shown above. Each s_i corresponds to either a single character or to a percent-escape triplet consisting of a `%` character followed by two hexadecimal digit characters.
2. Arrange the resulting digits and characters as shown for the GS1 Element String. If any s_i in the EPC URI is a percent-escape triplet `%xx`, in the GS1 Element String replace the triplet with the corresponding character according to Table 46 (Appendix A). (For a given percent-escape triplet `%xx`, find the row of Table 46 that contains `xx` in the "Hex Value" column; the "Graphic Symbol" column then gives the corresponding character to use in the GS1 Element String.)

To find the EPC URI corresponding to a GS1 element string that includes a GIAI (AI 8004):

1. Number the digits and characters of the GS1 element string as shown above.
2. Determine the number of digits *L* in the GS1 Company Prefix. This may be done, for example, by reference to an external table of company prefixes.
3. Arrange the digits as shown for the EPC URI. For each serial number character s_i, replace it with the corresponding value in the "URI Form" column of Table 46 (Appendix A) – either the character itself or a percent-escape triplet if s_i is not a legal URI character.

EPC URI: `urn:epc:id:giai:0614141.32a%2Fb`

GS1 element string: `(8004) 0614141 32a/b`

Spaces have been added to the GS1 element string for clarity, but they are never encoded. In this example, the slash (`/`) character in the serial number must be represented as an escape triplet in the EPC URI.

7.6 Global Service Relation Number (GSRN)

The GSRN EPC (Section 6.3.6) corresponds directly to the GSRN key defined in Sections 2.5 and 3.9.9 of the GS1 General Specifications [GS1GS10.0].

The correspondence between the GSRN EPC URI and a GS1 element string consisting of a GSRN key (AI 8018) is depicted graphically below:

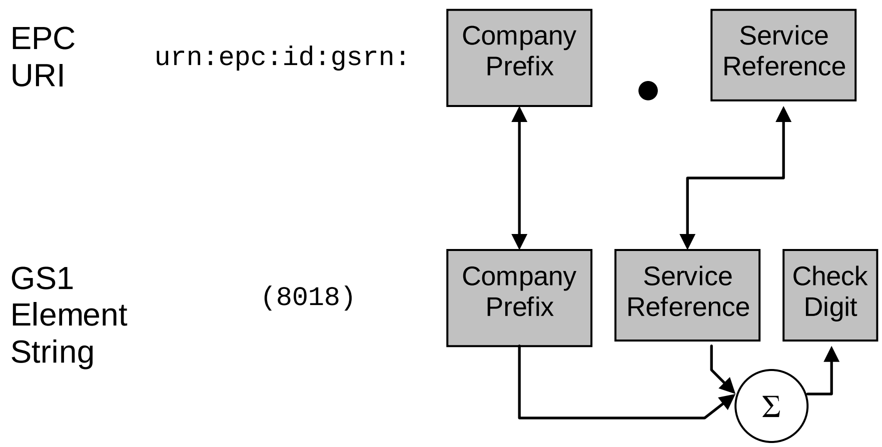


Figure 13.Correspondence between GSRN EPC URI and GS1 Element String

Formally, the correspondence is defined as follows. Let the EPC URI and the GS1 element string be written as follows:

EPC URI: `urn:epc:id:gsrn:`$d_1d_2…d_L.d_{(L+2)}d_{(L+3)}…d_{17}$

GS1 Element String: `(8018)`$d_1d_2…d_{18}$

To find the GS1 element string corresponding to a GSRN EPC URI:

1. Number the digits of the two components of the EPC as shown above. Note that there will always be a total of 17 digits.
2. Calculate the check digit $d_{18} = (10 – ((3(d_1 + d_3 + d_5 + d_7 + d_9 + d_{11} + d_{13} + d_{15} + d_{17}) + (d_2 + d_4 + d_6 + d_8 + d_{10} + d_{12} + d_{14} + d_{16})) \bmod 10)) \bmod 10$.
3. Arrange the resulting digits and characters as shown for the GS1 Element String.

To find the EPC URI corresponding to a GS1 element string that includes a GSRN (AI 8018):

1. Number the digits and characters of the GS1 element string as shown above.
2. Determine the number of digits *L* in the GS1 Company Prefix. This may be done, for example, by reference to an external table of company prefixes.
3. Arrange the digits as shown for the EPC URI. Note that the GSRN check digit d_{18} is not included in the EPC URI.

Example:

EPC URI: `urn:epc:id:gsrn:0614141.1234567890`

GS1 element string: `(8018) 0614141 1234567890 2`

Spaces have been added to the GS1 element string for clarity, but they are never encoded.

7.7 Global Document Type Identifier (GDTI)

The GDTI EPC (Section 6.3.7) corresponds directly to a serialized GDTI key defined in Sections 2.6.13 and 3.5.9 of the GS1 General Specifications [GS1GS10.0]. Because an EPC always identifies a specific physical object, only GDTI keys that include the optional serial number have a corresponding GDTI EPC. GDTI keys that lack a serial number refer to document classes rather than specific documents, and therefore do not have a corresponding EPC (just as a GTIN key without a serial number does not have a corresponding EPC).

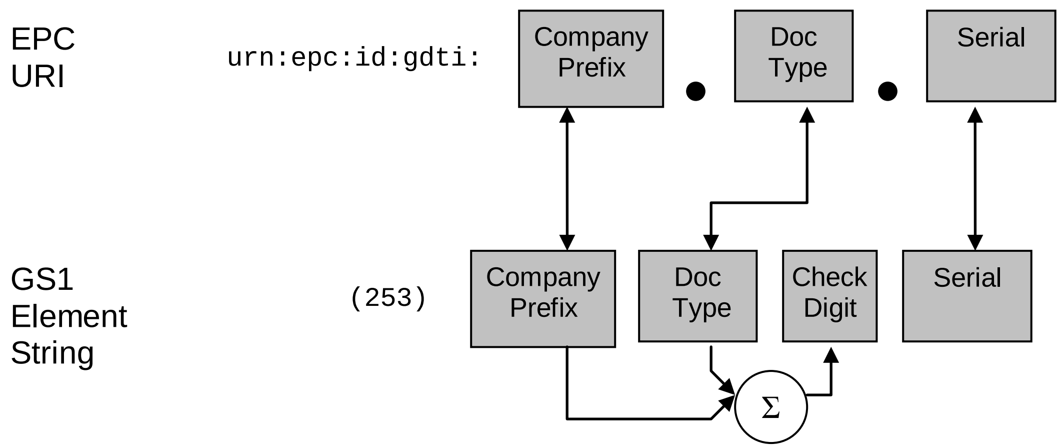


Figure 14.Correspondence between GDTI EPC URI and GS1 Element String

Formally, the correspondence is defined as follows. Let the EPC URI and the GS1 element string be written as follows:

EPC URI: `urn:epc:id:gdti:`$d_1 d_2 \ldots d_L . d_{(L+1)} d_{(L+2)} \ldots d_{12} . s_1 s_2 \ldots s_K$

GS1 Element String: `(253)`$d_1 d_2 \ldots d_{13} s_1 s_2 \ldots s_K$

To find the GS1 element string corresponding to a GRAI EPC URI:

1. Number the digits of the first two components of the EPC as shown above. Note that there will always be a total of 12 digits.
2. Number the characters of the serial number (third) component of the EPC as shown above. Each s_i is a digit character.
3. Calculate the check digit $d_{13} = (10 – ((3(d_2 + d_4 + d_6 + d_8 + d_{10} + d_{12}) + (d_1 + d_3 + d_5 + d_7 + d_9 + d_{11})) \bmod 10)) \bmod 10$.
4. Arrange the resulting digits as shown for the GS1 Element String.

To find the EPC URI corresponding to a GS1 element string that includes a GDTI (AI 253):

1. If the number of characters following the `(253)` application identifier is less than or equal to 13, stop: this element string does not have a corresponding EPC because it does not include the optional serial number.

2. Number the digits and characters of the GS1 element string as shown above.
3. Determine the number of digits *L* in the GS1 Company Prefix. This may be done, for example, by reference to an external table of company prefixes.
4. Arrange the digits as shown for the EPC URI. Note that the GDTI check digit d_{13} is not included in the EPC URI.

Example:

EPC URI: `urn:epc:id:gdti:0614141.12345.006847`

GS1 element string: `(253) 0 0614141 12345 2 006847`

Spaces have been added to the GS1 element string for clarity, but they are never encoded.

8 URIs for EPC Pure Identity Patterns

Certain software applications need to specify rules for filtering lists of EPC pure identities according to various criteria. This specification provides a Pure Identity Pattern URI form for this purpose. A Pure Identity Pattern URI does not represent a single EPC, but rather refers to a set of EPCs. A typical Pure Identity Pattern URI looks like this:

`urn:epc:idpat:sgtin:0652642.*.*`

This pattern refers to any EPC SGTIN, whose GS1 Company Prefix is 0652642, and whose Item Reference and Serial Number may be anything at all. The tag length and filter bits are not considered at all in matching the pattern to EPCs.

In general, there is a Pure Identity Pattern URI scheme corresponding to each Pure Identity EPC URI scheme (Section 6.3), whose syntax is essentially identical except that any number of fields starting at the right may be a star (`*`). This is more restrictive than EPC Tag Pattern URIs (Section 13), in that the star characters must occupy adjacent rightmost fields and the range syntax is not allowed at all.

The pure identity pattern URI for the DoD Construct is as follows:

`urn:epc:idpat:usdod:`*`CAGECodeOrDODAACPat.serialNumberPat`*

with similar restrictions on the use of star (`*`).

8.1 Syntax

The grammar for Pure Identity Pattern URIs is given below.

```
IDPatURI ::= “urn:epc:idpat:” IDPatBody

IDPatBody ::= GIDIDPatURIBody | SGTINIDPatURIBody |
SGLNIDPatURIBody | GIAIIDPatURIBody | SSCCIDPatURIBody |
GRAIIDPatURIBody | GSRNIDPatURIBody | GDTIIDPatURIBody |
DODIDPatURI

GIDIDPatURIBody ::= “gid:” GIDIDPatURIMain

GIDIDPatURIMain ::=
    2*(NumericComponent “.”) NumericComponent
```

```
    | 2*(NumericComponent ".") "*"
    | NumericComponent ".*.*"
    | "*.*.*"

SGTINIDPatURIBody ::= "sgtin:" SGTINPatURIMain

SGTINPatURIMain ::=
      2*(PaddedNumericComponent ".") GS3A3Component
    | 2*(PaddedNumericComponent ".") "*"
    | PaddedNumericComponent ".*.*"
    | "*.*.*"

GRAIIDPatURIBody ::= "grai:" SGLNGRAIIDPatURIMain

SGLNIDPatURIBody ::= "sgln:" SGLNGRAIIDPatURIMain

SGLNGRAIIDPatURIMain ::=
      PaddedNumericComponent "."
PaddedNumericComponentOrEmpty "." GS3A3Component
    | PaddedNumericComponent "."
PaddedNumericComponentOrEmpty ".*"
    | PaddedNumericComponent ".*.*"
    | "*.*.*"

SCCIDPatURIBody ::= "sscc:" SSCCIDPatURIMain

SSCCIDPatURIMain ::=
      PaddedNumericComponent "." PaddedNumericComponent
    | PaddedNumericComponent ".*"
    | "*.*"

GIAIIDPatURIBody ::= "giai:" GIAIIDPatURIMain

GIAIIDPatURIMain ::=
      PaddedNumericComponent "." GS3A3Component
    | PaddedNumericComponent ".*"
    | "*.*"

GSRNIDPatURIBody ::= "gsrn:" GSRNIDPatURIMain

GSRNIDPatURIMain ::=
      PaddedNumericComponent "." PaddedNumericComponent
    | PaddedNumericComponent ".*"
    | "*.*"

GDTIIDPatURIBody ::= "gdti:" GDTIIDPatURIMain

GDTIIDPatURIMain ::=
      PaddedNumericComponent "."
PaddedNumericComponentOrEmpty "." PaddedNumericComponent
    | PaddedNumericComponent "."
PaddedNumericComponentOrEmpty ".*"
    | PaddedNumericComponent ".*.*"
    | "*.*.*"
```

```
DODIDPatURI ::= “urn:epc:idpat:usdod:” DODIDPatMain

DODIDPatMain ::=
     CAGECodeOrDODAAC “.” DoDSerialNumber
   | CAGECodeOrDODAAC “.*”
   | “*.*”
```

8.2 Semantics

The meaning of a Pure Identity Pattern URI (`urn:epc:idpat:`) is formally defined as denoting a set of a set of pure identity EPCs, respectively.

The set of EPCs denoted by a specific Pure Identity Pattern URI is defined by the following decision procedure, which says whether a given Pure Identity EPC URI belongs to the set denoted by the Pure Identity Pattern URI.

Let `urn:epc:idpat:`*`Scheme`*`:P1.P2...P`*`n`* be a Pure Identity Pattern URI. Let `urn:epc:id:`*`Scheme`*`:C1.C2...C`*`n`* be a Pure Identity EPC URI, where the *`Scheme`* field of both URIs is the same. The number of components (*`n`*) depends on the value of *`Scheme`*.

First, any Pure Identity EPC URI component `C`*`i`* is said to *match* the corresponding Pure Identity Pattern URI component `P`*`i`* if:

- `P`*`i`* is a `NumericComponent`, and `C`*`i`* is equal to `P`*`i`*; or
- `P`*`i`* is a `PaddedNumericComponent`, and `C`*`i`* is equal to `P`*`i`* both in numeric value as well as in length; or
- `P`*`i`* is a `GS3A3Component`, and `C`*`i`* is equal to `P`*`i`*, character for character; or
- `P`*`i`* is a `CAGECodeOrDODAAC`, and `C`*`i`* is equal to `P`*`i`*; or
- `P`*`i`* is a `StarComponent` (and `C`*`i`* is anything at all)

Then the Pure Identity EPC URI is a member of the set denoted by the Pure Identity Pattern URI if and only if `Ci` matches `Pi` for all $1 \leq i \leq n$.

9 Memory Organization of Gen 2 RFID Tags

9.1 Types of Tag Data

RFID Tags, particularly Gen 2 RFID Tags, may carry data of three different kinds:

- *Business Data* Information that describes the physical object to which the tag is affixed. This information includes the Electronic Product Code (EPC) that uniquely identifies the physical object, and may also include other data elements carried on the tag. This information is what business applications act upon, and so this data is commonly transferred between the data capture level and the business application level in a typical implementation architecture. Most standardized business data on an RFID tag is equivalent to business data that may be found in other data carriers, such as bar codes.

- *Control Information* Information that is used by data capture applications to help control the process of interacting with tags. Control Information includes data that helps a capturing application filter out tags from large populations to increase read efficiency, special handling information that affects the behavior of capturing application, information that controls tag security features, and so on. Control Information is typically *not* passed directly to business applications, though Control Information may influence how a capturing application presents business data to the business application level. Unlike Business Data, Control Information has no equivalent in bar codes or other data carriers.
- *Tag Manufacture Information* Information that describes the Tag itself, as opposed to the physical object to which the tag is affixed. Tag Manufacture information includes a manufacturer ID and a code that indicates the tag model. It may also include information that describes tag capabilities, as well as a unique serial number assigned at manufacture time. Usually, Tag Manufacture Information is like Control Information in that it is used by capture applications but not directly passed to business applications. In some applications, the unique serial number that may be a part of Tag Manufacture Information is used in addition to the EPC, and so acts like Business Data. Like Control Information, Tag Manufacture Information has no equivalent in bar codes or other data carrriers.

It should be noted that these categories are slightly subjective, and the lines may be blurred in certain applications. However, they are useful for understanding how the Tag Data Standards are structured, and are a good guide for their effective and correct use.

The following table summarizes the information above.

Information Type	**Description**	**Where on Gen 2 Tag**	**Where Typically Used**	**Bar Code Equivalent**
Business Data	Describes the physical object to which the tag is affixed.	EPC Bank (excluding PC and XPC bits, and filter value within EPC) User Memory Bank	Data Capture layer and Business Application layer	Yes: GS1 keys, Application Identifiers (AIs)
Control Information	Facilitates efficient tag interaction	Reserved Bank EPC Bank: PC and XPC bits, and filter value within EPC	Data Capture layer	No

Information Type	Description	Where on Gen 2 Tag	Where Typically Used	Bar Code Equivalent
Tag Manufacture Information	Describes the tag itself, as opposed to the physical object to which the tag is affixed	TID Bank	Data Capture layer Unique tag manufacture serial number may reach Business Application layer	No

Table 3. Kinds of Data on a Gen 2 RFID Tag

9.2 Gen 2 Tag Memory Map

Binary data structures defined in the Tag Data Standard are intended for use in RFID Tags, particularly in UHF Class 1 Gen 2 Tags (also known as ISO 18000-6C Tags). The air interface standard [UHFC1G2] specifies the structure of memory on Gen 2 tags. Specifically, it specifies that memory in these tags consists of four separately addressable banks, numbered 00, 01, 10, and 11. It also specifies the intended use of each bank, and constraints upon the content of each bank dictated by the behavior of the air interface. For example, the layout and meaning of the Reserved bank (bank 00), which contains passwords that govern certain air interface commands, is fully specified in [UHFC1G2].

For those memory banks and memory locations that have no special meaning to the air interface (i.e., are "just data" as far as the air interface is concerned), the Tag Data Standard specifies the content and meaning of these memory locations.

Following the convention established in [UHFC1G2], memory addresses are described using hexadecimal bit addresses, where each bank begins with bit 00_h and extends upward to as many bits as each bank contains, the capacity of each bank being constrained in some respects by [UHFC1G2] but ultimately may vary with each tag make and model. Bit 00_h is considered the most significant bit of each bank, and when binary fields are laid out into tag memory the most significant bit of any given field occupies the lowest-numbered bit address occupied by that field. When describing individual fields, however, the least significant bit is numbered zero. For example, the Access Password is a 32-bit unsigned integer consisting of bits $b_{31}b_{30}\ldots b_0$, where b_{31} is the most significant bit and b_0 is the least significant bit. When the Access Password is stored at address 20_h – $3F_h$ (inclusive) in the Reserved bank of a Gen 2 tag, the most significant bit b_{31} is stored at tag address 20_h and the least significant bit b_0 is stored at address $3F_h$.

The following diagram shows the layout of memory on a Gen 2 tag, The colors indicate the type of data following the categorization in Section Figure 1.

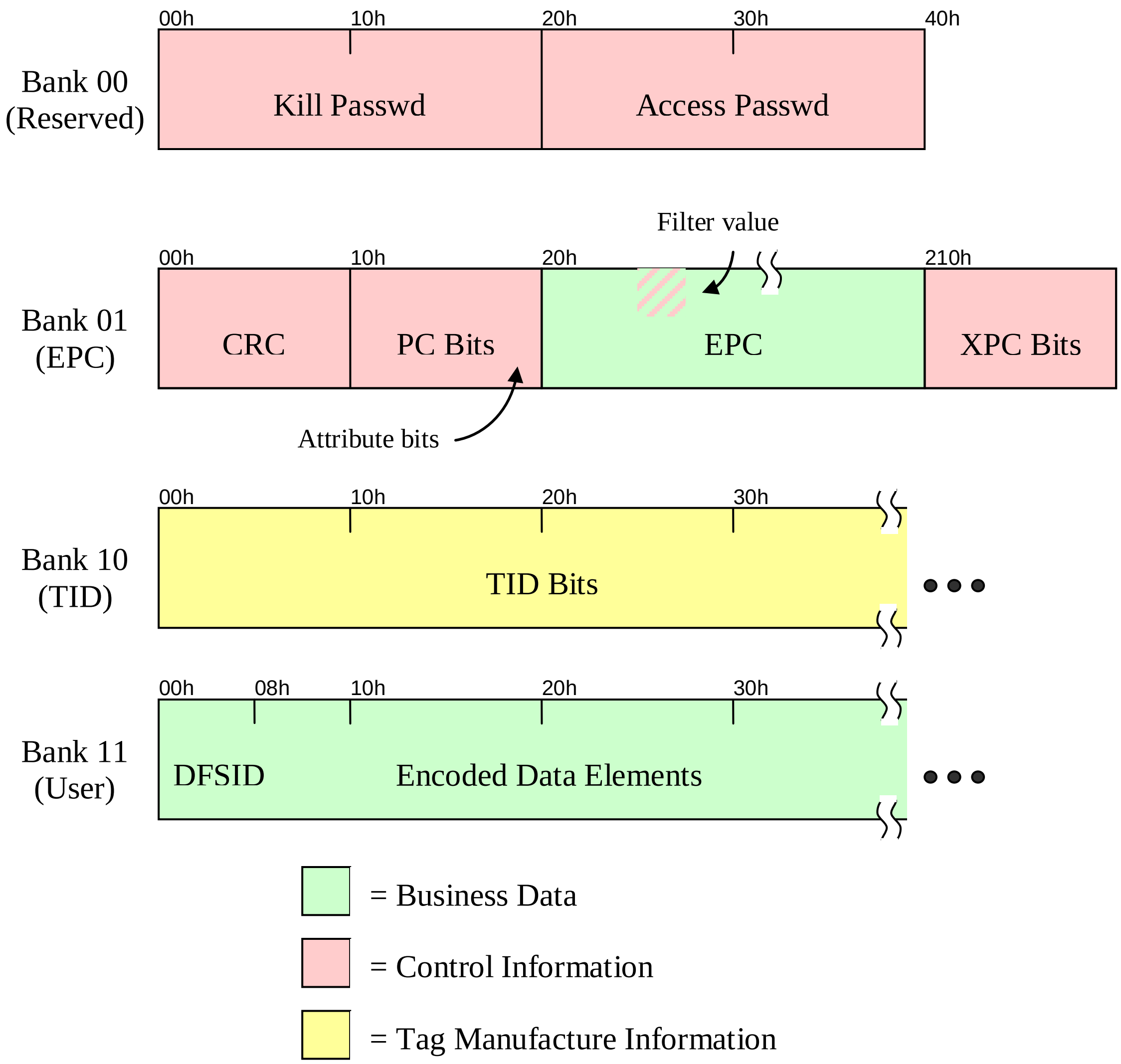


Figure 15.Gen 2 Tag Memory Map

The following table describes the fields in the memory map above.

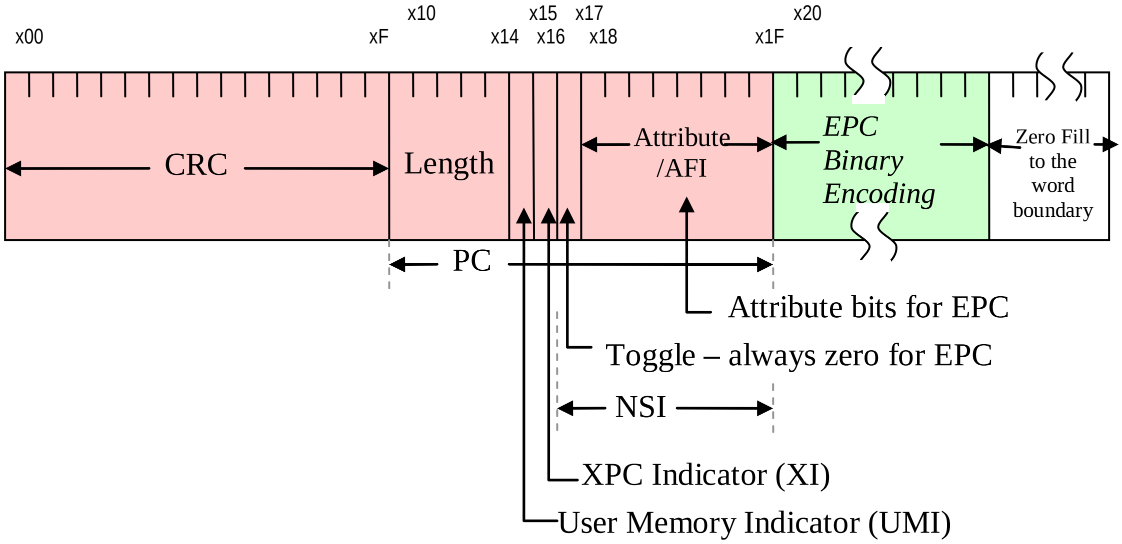
Bank	Bits	Field	Description	Category	Where Specified
Bank 00 (Reserved)	00_h – $1F_h$	Kill Passwd	A 32-bit password that must be presented to the tag in order to complete the Gen 2 “kill” command.	Control Info	[UHFC1G2]
	20_h – $2F_h$	Access Passwd	A 32-bit password that must be presented to the tag in order to perform privileged operations	Control Info	[UHFC1G2]

Bank	Bits	Field	Description	Category	Where Specified
Bank 01 (EPC)	00_h – $0F_h$	CRC	A 16-bit Cyclic Redundancy Check computed over the contents of the EPC bank.	Control Info	[UHFC1G2]
	10_h – $1F_h$	PC Bits	Protocol Control bits (see below)	Control Info	(see below)
	20_h – end	EPC	Electronic Product Code, plus filter value. The Electronic Product code is a globally unique identifier for the physical object to which the tag is affixed. The filter value provides a means to improve tag read efficiency by selecting a subset of tags of interest.	Business Data (except filter value, which is Control Info)	The EPC is defined in Sections 6, 7, and 13. The filter values are defined in Section 10.
	210_h – $21F_h$	XPC Bits	Extended Protocol Control bits. If bit 16_h of the EPC bank is set to one, then bits 210_h – $21F_h$ (inclusive) contain additional protocol control bits as specified in [UHFC1G2]	Control Info	[UHFC1G2]
Bank 10 (TID)	00_h – end	TID Bits	Tag Identification bits, which provide information about the tag itself, as opposed to the physical object to which the tag is affixed.	Tag Manu-facture Info	Section 16

Bank	Bits	Field	Description	Category	Where Specified
Bank 11 (User)	00_h – end	DSFID	Logically, the content of user memory is a set of name-value pairs, where the name part is an OID [ASN.1] and the value is a character string. Physically, the first few bits are a Data Storage Format Identifier as specified in [ISO15961] and [ISO15962]. The DSFID specifies the format for the remainder of the user memory bank. The DSFID is typically eight bits in length, but may be extended further as specified in [ISO15961]. When the DSFID specifies Access Method 2, the format of the remainder of user memory is "packed objects" as specified in Section 17. This format is recommended for use in EPC applications. The physical encoding in the packed objects data format is as a sequence of "packed objects," where each packed object includes one or more name-value pairs whose values are compacted together.	Business Data	[ISO15961], [ISO15962], Section 17

Table 4. Gen 2 Memory Map

The following diagram illustrates in greater detail the first few bits of the EPC Bank (Bank 01), and in particular shows the various fields within the Protocol Control bits (bits 10_h – $1F_h$, inclusive).



1625

1626 Figure 16.Gen 2 Protocol Control (PC) Bits Memory Map

1627 The following table specifies the meaning of the PC bits:

Bits	Field	Description	Where Specified
10_h – 14_h	Length	Represents the number of 16-bit words comprising the PC field and the EPC field (below). See discussion below for the encoding of this field.	[UHFC1G2]
15_h	User Memory Indicator (UMI)	Indicates whether the user memory bank is present and contains data.	[UHFC1G2]
16_h	XPC Indicator (XI)	Indicates whether an XPC is present	[UHFC1G2]
17_h	Toggle	If zero, indicates an EPCglobal application; in particular, indicates that bits 18_h – $1F_h$ contain the Attribute Bits and the remainder of the EPC bank contains a binary encoded EPC. If one, indicates a non-EPCglobal application; in particular, indicates that bits 18_h – $1F_h$ contain the ISO Application Family Identifier (AFI) as defined in [ISO15961] and the remainder of the EPC bank contains a Unique Item Identifier (UII) appropriate for that AFI.	[UHFC1G2]

Bits	Field	Description	Where Specified
18_h – $1F_h$ (if toggle = 0)	Attribute Bits	Bits that may guide the handling of the physical object to which the tag is affixed.	Section 11
18_h – $1F_h$ (if toggle = 1)	AFI	An Application Family Identifier that specifies a non-EPCglobal application for which the remainder of the EPC bank is encoded	[ISO15961]

Table 5. Gen 2 Protocol Control (PC) Bits Memory Map

Bits 17_h – $1F_h$ (inclusive) are collectively known as the Numbering System Identifier (NSI). It should be noted, however, that when the toggle bit (bit 17_h) is zero, the numbering system is always the Electronic Product Code, and bits 18_h – $1F_h$ contain the Attribute Bits whose purpose is completely unrelated to identifying the numbering system being used.

10 Filter Value

The filter value is additional control information that may be included in the EPC memory bank of a Gen 2 tag. The intended use of the filter value is to allow an RFID reader to select or deselect the tags corresponding to certain physical objects, to make it easier to read the desired tags in an environment where there may be other tags present in the environment. For example, if the goal is to read the single tag on a pallet, and it is expected that there may be hundreds or thousands of item-level tags present, the performance of the capturing application may be improved by using the Gen 2 air interface to select the pallet tag and deselect the item-level tags.

Filter values are available for all EPC types except for the General Identifier (GID). There is a different set of standardized filter value values associated with each type of EPC, as specified below.

It is essential to understand that the filter value is additional "control information" that is *not* part of the Electronic Product Code. The filter value does not contribute to the unique identity of the EPC. For example, it is *not* permissible to attach two RFID tags to to different physical objects where both tags contain the same EPC, even if the filter values are different on the two tags.

Because the filter value is not part of the EPC, the filter value is *not* included when the EPC is represented as a pure identity URI, nor should the filter value be considered as part of the EPC by business applications. Capturing applications may, however, read the filter value and pass it upwards to business applications in some data field other than the EPC. It should be recognized, however, that the purpose of the filter values is to assist in the data capture process, and in most cases the filter value will be of limited or no value to business applications. The filter value is *not* intended to provide a reliable packaging-level indicator for business applications to use.

10.1 Use of "Reserved" and "All Others" Filter Values

In the following sections, filter values marked as "reserved" are reserved for assignment by EPCglobal in future versions of this specification. Implementations of the encoding and decoding rules specified herein SHALL accept any value of the filter values, whether reserved or not. Applications, however, SHOULD NOT direct an encoder to write a reserved value to a tag, nor rely upon a reserved value decoded from a tag, as doing so may cause interoperability problems if a reserved value is assigned in a future revision to this specification.

Each EPC scheme includes a filter value identified as "All Others." This filter value means that the object to which the tag is affixed does not match the description of any of the other filter values defined for that EPC scheme. In some cases, the "All Others" filter value may appear on a tag that was encoded to conform to an earlier version of this specification, at which time no other suitable filter value was available. When encoding a new tag, the filter value should be set to match the description of the object to which the tag is affixed, with "All Others" being used only if a suitable filter value for the object is not defined in this specification.

10.2 Filter Values for SGTIN EPC Tags

The normative specifications for Filter Values for SGTIN EPC Tags are specified below.

Type	Filter Value	Binary Value
All Others (see Section 10.1)	0	000
Point of Sale (POS) Trade Item	1	001
Full Case for Transport	2	010
Reserved (see Section 10.1)	3	011
Inner Pack Trade Item Grouping for Handling	4	100
Reserved (see Section 10.1)	5	101
Unit Load (see Section 10.1)	6	110
Unit inside Trade Item or component inside a product not intended for individual sale	7	111

Table 6. SGTIN Filter Values

10.3 Filter Values for SSCC EPC Tags

The normative specifications for Filter Values for SSCC EPC Tags are specified below.

Type	Filter Value	Binary Value
All Others (see Section 10.1)	0	000

Type	Filter Value	Binary Value
Reserved (see Section 10.1)	1	001
Full Case for Transport	2	010
Reserved (see Section 10.1)	3	011
Reserved (see Section 10.1)	4	100
Reserved (see Section 10.1)	5	101
Unit Load (see Section 10.1)	6	110
Reserved (see Section 10.1)	7	111

Table 7. SSCC Filter Values

10.4 Filter Values for SGLN EPC Tags

Type	Filter Value	Binary Value
All Others (see Section 10.1)	0	000
Reserved (see Section 10.1)	1	001
Reserved (see Section 10.1)	2	010
Reserved (see Section 10.1)	3	011
Reserved (see Section 10.1)	4	100
Reserved (see Section 10.1)	5	101
Reserved (see Section 10.1)	6	110
Reserved (see Section 10.1)	7	111

Table 8. SGLN Filter Values

10.5 Filter Values for GRAI EPC Tags

Type	Filter Value	Binary Value
All Others (see Section 10.1)	0	000
Reserved (see Section 10.1)	1	001
Reserved (see Section 10.1)	2	010
Reserved (see Section 10.1)	3	011
Reserved (see Section 10.1)	4	100
Reserved (see Section 10.1)	5	101
Reserved (see Section 10.1)	6	110
Reserved (see Section 10.1)	7	111

Table 9. GRAI Filter Values

10.6 Filter Values for GIAI EPC Tags

Type	Filter Value	Binary Value
All Others (see Section 10.1)	0	000
Reserved (see Section 10.1)	1	001
Reserved (see Section 10.1)	2	010
Reserved (see Section 10.1)	3	011
Reserved (see Section 10.1)	4	100
Reserved (see Section 10.1)	5	101
Reserved (see Section 10.1)	6	110
Reserved (see Section 10.1)	7	111

Table 10. GIAI Filter Values

10.7 Filter Values for GSRN EPC Tags

Type	Filter Value	Binary Value
All Others (see Section 10.1)	0	000
Reserved (see Section 10.1)	1	001
Reserved (see Section 10.1)	2	010
Reserved (see Section 10.1)	3	011
Reserved (see Section 10.1)	4	100
Reserved (see Section 10.1)	5	101
Reserved (see Section 10.1)	6	110
Reserved (see Section 10.1)	7	111

Table 11. GSRN Filter Values

10.8 Filter Values for GDTI EPC Tags

Type	Filter Value	Binary Value
All Others (see Section 10.1)	0	000
Reserved (see Section 10.1)	1	001
Reserved (see Section 10.1)	2	010
Reserved (see Section 10.1)	3	011
Reserved (see Section 10.1)	4	100

Type	Filter Value	Binary Value
Reserved (see Section 10.1)	5	101
Reserved (see Section 10.1)	6	110
Reserved (see Section 10.1)	7	111

Table 12. GDTI Filter Values

10.9 Filter Values for GID EPC Tags

The GID EPC scheme does not provide for the use of filter values.

10.10 Filter Values for DOD EPC Tags

Filter values for US DoD EPC Tags are as specified in [USDOD].

11 Attribute Bits

The Attribute Bits are eight bits of "control information" that may be used by capturing applications to guide the capture process. Attribute Bits may be used to determine whether the physical object to which a tag is affixed requires special handling of any kind.

Attribute bits are available for all EPC types. The same definitions of attribute bits as specified below apply regardless of which EPC scheme is used.

It is essential to understand that attribute bits are additional "control information" that is *not* part of the Electronic Product Code. Attribute bits do not contribute to the unique identity of the EPC. For example, it is *not* permissible to attach two RFID tags to different physical objects where both tags contain the same EPC, even if the attribute bits are different on the two tags.

Because attribute bits are not part of the EPC, they are *not* included when the EPC is represented as a pure identity URI, nor should the attribute bits be considered as part of the EPC by business applications. Capturing applications may, however, read the attribute bits and pass them upwards to business applications in some data field other than the EPC. It should be recognized, however, that the purpose of the attribute bits is to assist in the data capture and physical handling process, and in most cases the attribute bits will be of limited or no value to business applications. The attribute bits are *not* intended to provide a reliable master data or product descriptive attributes for business applications to use.

The currently assigned attribute bits are as specified below:

Bit Address	Assigned as of TDS Version	Meaning
18_h	[unassigned]	
19_h	[unassigned]	

Bit Address	Assigned as of TDS Version	Meaning
$1A_h$	[unassigned]	
$1B_h$	[unassigned]	
$1C_h$	[unassigned]	
$1D_h$	[unassigned]	
$1E_h$	[unassigned]	
$1F_h$	1.5	A “1” bit indicates the tag is affixed to hazardous material. A “0” bit provides no such indication.

Table 13. Attribute Bit Assignments

In the table above, attribute bits marked as “unassigned” are reserved for assignment by EPCglobal in future versions of this specification. Implementations of the encoding and decoding rules specified herein SHALL accept any value of the attribute bits, whether reserved or not. Applications, however, SHOULD direct an encoder to write a zero for each unassigned bit, and SHOULD NOT rely upon the value of an unassigned bit decoded from a tag, as doing so may cause interoperability problems if an unassigned value is assigned in a future revision to this specification.

12 EPC Tag URI and EPC Raw URI

The EPC memory bank of a Gen 2 tag contains a binary-encoded EPC, along with other control information. Applications do not normally process binary data directly. An application wishing to read the EPC may receive the EPC as a Pure Identity EPC URI, as defined in Section 6. In other situations, however, a capturing application may be interested in the control information on the tag as well as the EPC. Also, an application that writes the EPC memory bank needs to specify the values for control information that are written along with the EPC. In both of these situations, the EPC Tag URI and EPC Raw URI may be used.

The EPC Tag URI specifies both the EPC and the values of control information in the EPC memory bank. It also specifies which of several variant binary coding schemes is to be used (e.g., the choice between SGTIN-96 and SGTIN-198). As such, an EPC Tag URI completely and uniquely specifies the contents of the EPC memory bank. The EPC Raw URI also specifies the complete contents of the EPC memory bank, but repesents the memory contents as a single decimal or hexadecimal numeral.

12.1 Structure of the EPC Tag URI and EPC Raw URI

The EPC Tag URI begins with `urn:epc:tag:`, and is used when the EPC memory bank contains a valid EPC. EPC Tag URIs resemble Pure Identity EPC URIs, but with added control information. The EPC Raw URI begins with `urn:epc:raw:`, and is used when the EPC memory bank does not contain a valid EPC. This includes situations where the toggle bit (bit 17_h) is set to one, as well as situations where the toggle bit is set to zero but the remainder of the EPC bank does not conform to the coding rules specified in Section 14, either because the header bits are unassigned or the remainder of the binary encoding violates a validity check for that header.

The following figure illustrates these URI forms.

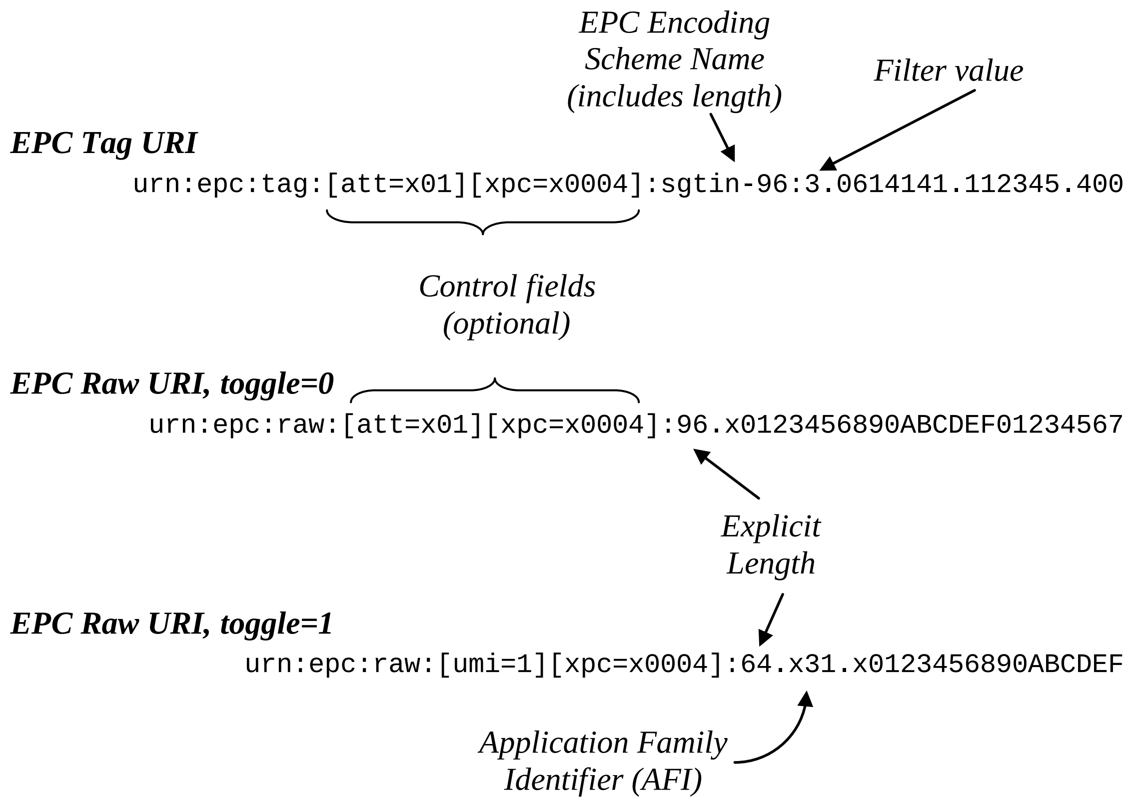


Figure 17.Illustration of EPC Tag URI and EPC Raw URI

The first form in the figure, the EPC Tag URI, is used for a valid EPC. It resembles the Pure Identity EPC URI, with the addition of optional control information fields as specified in Section 12.2.2 and a (non-optional) filter value. The EPC scheme name (`sgtin-96` in the example above) specifies a particular binary encoding scheme, and so it includes the length of the encoding. This is in contrast to the Pure Identity EPC URI which identifies an EPC scheme but not a specific binary encoding (e.g., `sgtin` but not specifically `sgtin-96`).

The EPC Raw URI illustrated by the second example in the figure can be used whenever the toggle bit (bit 17_h) is zero, but is typically only used if the first form cannot (that is, if the contents of the EPC bank cannot be decoded according to Section 14.4). It specifies the contents of bit 20_h onward as a single hexadecimal numeral. The number of bits in this numeral is determined by the "length" field in the EPC bank of the tag (bits 10_h – 14_h). (The grammar in Section 12.4 includes a variant of this form in which the contents are specified as a decimal numeral. This form is deprecated.)

The EPC Raw URI illustrated by the third example in the figure is used when the toggle bit (bit 17_h) is one. It is similar to the second form, but with an additional field between the length and payload that reports the value of the AFI field (bits 18_h – $1F_h$) as a hexadecimal numeral.

Each of these forms is fully defined by the encoding and decoding procedures specified in Section 15.

12.2 Control Information

The EPC Tag URI and EPC Raw URI specify the complete contents of the Gen 2 EPC memory bank, including control information such as filter values and attribute bits. This section specifies how control information is included in these URIs.

12.2.1 Filter Values

Filter values are only available when the EPC bank contains a valid EPC, and only then when the EPC is an EPC scheme other than GID. In the EPC Tag URI, the filter value is indicated as an additional field following the scheme name and preceding the remainder of the EPC, as illustrated below:

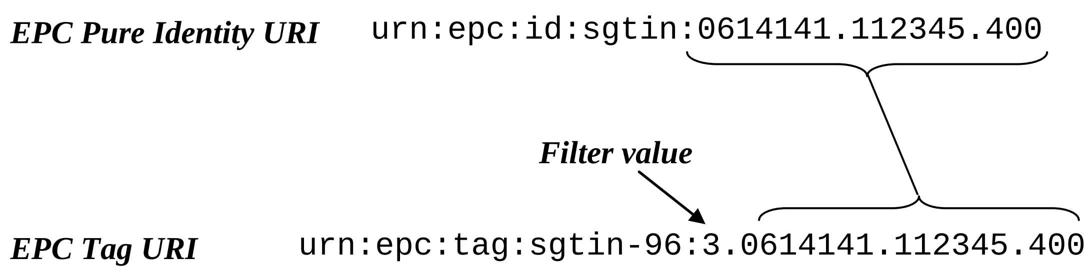


Figure 18.Illustration of Filter Value Within EPC Tag URI

The filter value is a decimal integer. The allowed values of the filter value are specified in Section 10.

12.2.2 Other Control Information Fields

Control information in the EPC bank apart from the filter values is stored separately from the EPC. Such information can be represented both in the EPC Tag URI and the EPC Raw URI, using the name-value pair syntax described below.

In both URI forms, control field name-value pairs may occur following the `urn:epc:tag:` or `urn:epc:raw:`, as illustrated below:

```
urn:epc:tag:[att=x01][xpc=x0004]:sgtin-96:3.0614141.112345.400
urn:epc:raw:[att=x01][xpc=x0004]:96.x012345689ABCDEF01234567
```

Each element in square brackets specifies the value of one control information field. An omitted field is equivalent to specifying a value of zero. As a limiting case, if no control information fields are specified in the URI it is equivalent to specifying a value of zero for all fields. This provides back-compatibility with earlier versions of the Tag Data Standard.

The available control information fields are specified in the following table.

Field	Syntax	Description	Read/Write
Attribute Bits	`[att=xNN]`	The value of the attribute bits (bits 18_h – $1F_h$), as a two-digit hexadecimal numeral *NN*. This field is only available if the toggle bit (bit 17_h) is zero.	Read / Write
User Memory Indicator	`[umi=B]`	The value of the user memory indicator bit (bit 15_h). The value *B* is either the digit 0 or the digit 1.	Read / Write Note that certain Gen 2 Tags may ignore the value written to this bit, and instead calculate the value of the bit from the contents of user memory. See [UHFC1G2].
Extended PC Bits	`[xpc=xNNNN]`	The value of the XPC bits (bits 210_h-$21F_h$) as a four-digit hexadecimal numeral *NNNN*.	Read only

Table 14. Control Information Fields

The user memory indicator and extended PC bits are calculated by the tag as a function of other information on the tag or based on operations performed to the tag (such as recommissioning). Therefore, these fields cannot be written directly. When reading from a tag, any of the control information fields may appear in the URI that results from decoding the EPC memory bank. When writing a tag, the `umi` and `xpc` fields will be ignored when encoding the URI into the tag.

To aid in decoding, any control information fields that appear in a URI must occur in alphabetical order (the same order as in the table above).

Examples (non-normative): The following examples illustrate the use of control information fields in the EPC Tag URI and EPC Raw URI.

`urn:epc:tag:sgtin-96:3.0614141.112345.400`

This is a tag with an SGTIN EPC, filter bits = 3, the hazardous material attribute bit set to zero, no user memory (user memory indicator = 0), and not recommissioned (extended PC = 0). This illustrates back-compatibility with earlier versions of the Tag Data Standard.

`urn:epc:tag:[att=x01]:sgtin-96:3.0614141.112345.400`

This is a tag with an SGTIN EPC, filter bits = 3, the hazardous material attribute bit set to one, no user memory (user memory indicator = 0), and not recommissioned (extended PC = 0). This URI might be specified by an application wishing to commission a tag with the hazardous material bit set to one and the filter bits and EPC as shown.

`urn:epc:raw:[att=x01][umi=1][xpc=x0004]:96.x1234567890ABCDEF01234567`

This is a tag with toggle=0, random data in bits 20_h *onward (not decodable as an EPC), the hazardous material attribute bit set to one, non-zero contents in user memory, and has been recommissioned (as indicated by the extended PC).*

`urn:epc:raw:[xpc=x0001]:96.xC1.x1234567890ABCDEF01234567`

This is a tag with toggle=1, Application Family Indicator = C1 (hexadecimal), and has had its user memory killed (as indicated by the extended PC).

12.3 EPC Tag URI and EPC Pure Identity URI

The Pure Identity EPC URI as defined in Section 6 is a representation of an EPC for use in information systems. The only information in a Pure Identity EPC URI is the EPC itself. The EPC Tag URI, in contrast, contains additional information: it specifies the contents of all control information fields in the EPC memory bank, and it also specifies which encoding scheme is used to encode the EPC into binary. Therefore, to convert a Pure Identity EPC URI to an EPC Tag URI, additional information must be provided. Conversely, to extract a Pure Identity EPC URI from an EPC Tag URI, this additional information is removed. The procedures in this section specify how these conversions are done.

12.3.1 EPC Binary Coding Schemes

For each EPC scheme as specified in Section 6, there are one or more corresponding EPC Binary Coding Schemes that determine how the EPC is encoded into binary representation for use in RFID tags. When there is more than one EPC Binary Coding Scheme available for a given EPC scheme, a user must choose which binary coding scheme to use. In general, the shorter binary coding schemes result in fewer bits and therefore permit the use of less expensive RFID tags containing less memory, but are restricted in the range of serial numbers that are permitted. The longer binary coding schemes allow for the full range of serial numbers permitted by the GS1 General Specifications, but require more bits and therefore more expensive RFID tags.

It is important to note that two EPCs are the same if and only if the Pure Identity EPC URIs are character for character identical. A long binary encoding (e.g., SGTIN-198) is *not* a different EPC from a short binary encoding (e.g., SGTIN-96) if the GS1 Company Prefix, item reference with indicator, and serial numbers are identical.

The following table enumerates the available EPC binary coding schemes, and indicates the limitations imposed on serial numbers.

EPC Scheme	EPC Binary Coding Scheme	EPC + Filter Bit Count	Includes Filter Value	Serial Number Limitation
`sgtin`	`sgtin-96`	96	Yes	Numeric-only, no leading zeros, decimal value must be less than 2^{38} (i.e., decimal value less than or equal to 274,877,906,943).
	`sgtin-198`	198	Yes	All values permitted by GS1 General Specifications (up to 20 alphanumeric characters)
`sscc`	`sscc-96`	96	Yes	All values permitted by GS1 General Specifications (11 – 5 decimal digits including extension digit, depending on GS1 Company Prefix length)
`sgln`	`sgln-96`	96	Yes	Numeric-only, no leading zeros, decimal value must be less than 2^{41} (i.e., decimal value less than or equal to 2,199,023,255,551).
	`sgln-195`	195	Yes	All values permitted by GS1 General Specifications (up to 20 alphanumeric characters)
`grai`	`grai-96`	96	Yes	Numeric-only, no leading zeros, decimal value must be less than 2^{38} (i.e., decimal value less than or equal to 274,877,906,943).
	`grai-170`	170	Yes	All values permitted by GS1 General Specifications (up to 16 alphanumeric characters)
`giai`	`giai-96`	96	Yes	Numeric-only, no leading zeros, decimal value must be less than a limit that varies according to the length of the GS1 Company Prefix. See Section 14.5.5.1.
	`giai-202`	202	Yes	All values permitted by GS1 General Specifications (up to 18 – 24 alphanumeric characters, depending on company prefix length)
`gsrn`	`gsrn-96`	96	Yes	All values permitted by GS1 General Specifications (11 – 5 decimal digits, depending on GS1 Company Prefix length)

<table>
<tr><th>EPC Scheme</th><th>EPC Binary Coding Scheme</th><th>EPC + Filter Bit Count</th><th>Includes Filter Value</th><th>Serial Number Limitation</th></tr>
<tr><td rowspan="2"><code>gdti</code></td><td><code>gdti-96</code></td><td>96</td><td>Yes</td><td>Numeric-only, no leading zeros, decimal value must be less than 2^{41} (i.e., decimal value less than or equal to 2,199,023,255,551).</td></tr>
<tr><td><code>gdti-113</code></td><td>113</td><td>Yes</td><td>All values permitted by GS1 General Specifications (up to 17 decimal digits, with or without leading zeros)</td></tr>
<tr><td><code>gid</code></td><td><code>gid-96</code></td><td>96</td><td>No</td><td>Numeric-only, no leading zeros, decimal value must be less than 2^{36} (i.e., decimal value must be less than or equal to 68,719,476,735).</td></tr>
<tr><td><code>usdod</code></td><td><code>usdod-96</code></td><td>96</td><td colspan="2">See “United States Department of Defense Supplier's Passive RFID Information Guide” that can be obtained at the United States Department of Defense's web site (http://www.dodrfid.org/supplierguide.htm).</td></tr>
</table>

Table 15. EPC Binary Coding Schemes and Their Limitations

Explanation (non-normative): For the SGTIN, SGLN, GRAI, and GIAI EPC schemes, the serial number according to the GS1 General Specifications is a variable length, alphanumeric string. This means that serial number `34`, `034`, `0034`, *etc, are all different serial numbers, as are* `P34`, `34P`, `0P34`, `P034`, *and so forth. In order to provide for up to 20 alphanumeric characters, 140 bits are required to encode the serial number. This is why the “long” binary encodings all have such a large number of bits. Similar considerations apply to the GDTI EPC scheme, except that the GDTI only allows digit characters (but still permits leading zeros).*

In order to accommodate the very common 96-bit RFID tag, additional binary coding schemes are introduced that only require 96 bits. In order to fit within 96 bits, some serial numbers have to be excluded. The 96-bit encodings of SGTIN, SGLN, GRAI, GIAI, and GDTI are limited to serial numbers that consist only of digits, which do not have leading zeros (unless the serial number consists in its entirety of a single `0` *digit), and whose value when considered as a decimal numeral is less than 2^B, where B is the number of bits available in the binary coding scheme. The choice to exclude serial numbers with leading zeros was an arbitrary design choice at the time the 96-bit encodings were first defined; for example, an alternative would have been to permit leading zeros, at the expense of excluding other serial numbers. But it is impossible to escape the fact that in B bits there can be no more than 2^B different serial numbers.*

When decoding a “long” binary encoding, it is not *permissible to strip off leading zeros when the binary encoding includes leading zero characters. Likewise, when encoding an EPC into either the “short” or “long” form, it is not permissible to strip off leading zeros* *prior to encoding. This means that EPCs whose serial numbers have leading zeros can only be encoded in the "long" form.*

In certain applications, it is desirable for the serial number to always contain a specific number of characters. Reasons for this may include wanting a predictable length for the EPC URI string, or for having a predictable size for a corresponding bar code encoding of the same identifier. In certain bar code applications, this is accomplished through the use of leading zeros. If 96-bit tags are used, however, the option to use leading zeros does not exist.

Therefore, in applications that both require 96-bit tags and *require that the serial number be a fixed number of characters, it is recommended that numeric serial numbers be used that are in the range* $10^D \le serial < 10^{D+1}$*, where D is the desired number of digits. For example, if 11-digit serial numbers are desired, an application can use serial numbers in the range 10,000,000,000 through 99,999,999,999. Such applications must take care to use serial numbers that fit within the constraints of 96-bit tags. For example, if 12-digit serial numbers are desired for SGTIN-96 encodings, then the serial numbers must be in the range 100,000,000,000 through 274,877,906,943.*

It should be remembered, however, that many applications do not require a fixed number of characters in the serial number, and so all serial numbers from 0 through the maximum value (without leading zeros) may be used with 96-bit tags.

12.3.2 EPC Pure Identity URI to EPC Tag URI

Given:

- An EPC Pure Identity URI as specified in Section 6. This is a string that matches the `EPC-URI` production of the grammar in Section 6.3.
- A selection of a binary coding scheme to use. This is one of the the binary coding schemes specified in the "EPC Binary Coding Scheme" column of Table 15. The chosen binary coding scheme must be one that corresponds to the EPC scheme in the EPC Pure Identity URI.
- A filter value, if the "Includes Filter Value" column of Table 15 indicates that the binary encoding includes a filter value.
- The value of the attribute bits.
- The value of the user memory indicator.

Validation:

- The serial number portion of the EPC (the characters following the rightmost dot character) must conform to any restrictions implied by the selected binary coding scheme, as specified by the "Serial Number Limitation" column of Table 15.
- The filter value must be in the range $0 \le filter \le 7$.

Procedure:

1. Starting with the EPC Pure Identity URI, replace the prefix `urn:epc:id:` with `urn:epc:tag:`.
2. Replace the EPC scheme name with the selected EPC binary coding scheme name. For example, replace `sgtin` with `sgtin-96` or `sgtin-198`.
3. If the selected binary coding scheme includes a filter value, insert the filter value as a single decimal digit following the rightmost colon (“`:`”) character of the URI, followed by a dot (“`.`”) character.
4. If the attribute bits are non-zero, construct a string `[att=x`*`NN`*`]`, where *`NN`* is the value of the attribute bits as a 2-digit hexadecimal numeral.
5. If the user memory indicator is non-zero, construct a string `[umi=1]`.
6. If Step 4 or Step 5 yielded a non-empty string, insert those strings following the rightmost colon (“`:`”) character of the URI, followed by an additional colon character.
7. The resulting string is the EPC Tag URI.

12.3.3 EPC Tag URI to EPC Pure Identity URI

Given:

- An EPC Tag URI as specified in Section 12. This is a string that matches the `TagURI` production of the grammar in Section 12.4.

Procedure:

1. Starting with the EPC Tag URI, replace the prefix `urn:epc:tag:` with `urn:epc:id:`.
2. Replace the EPC binary coding scheme name with the corresponding EPC scheme name. For example, replace `sgtin-96` or `sgtin-198` with `sgtin`.
3. If the coding scheme includes a filter value, remove the filter value (the digit following the rightmost colon character) and the following dot (“`.`”) character.
4. If the URI contains one or more control fields as specified in Section 12.2.2, remove them and the following colon character.
5. The resulting string is the Pure Identity EPC URI.

12.4 Grammar

The following grammar specifies the syntax of the EPC Tag URI and EPC Raw URI. The grammar makes reference to grammatical elements defined in Sections 5 and 6.3.

```
TagOrRawURI ::= TagURI | RawURI

TagURI ::=  “urn:epc:tag:” TagURIControlBody

TagURIControlBody ::= ( ControlField+ “:” )? TagURIBody
```

```
TagURIBody ::= SGTINTagURIBody | SSCCTagURIBody |
SGLNTagURIBody | GRAITagURIBody | GIAITagURIBody |
GDTITagURIBody | GSRNTagURIBody | GIDTagURIBody |
DODTagURIBody

SGTINTagURIBody ::= SGTINEncName “:” NumericComponent “.”
SGTINURIBody

SGTINEncName ::= “sgtin-96” | “sgtin-198”

SSCCTagURIBody ::= SSCCEncName “:” NumericComponent “.”
SSCCURIBody

SSCCEncName ::= “sscc-96”

SGLNTagURIBody ::= SGLNEncName “:” NumericComponent “.”
SGLNURIBody

SGLNEncName ::= “sgln-96” | “sgln-195”

GRAITagURIBody ::= GRAIEncName “:” NumericComponent “.”
GRAIURIBody

GRAIEncName ::= “grai-96” | “grai-170”

GIAITagURIBody ::= GIAIEncName “:” NumericComponent “.”
GIAIURIBody

GIAIEncName ::= “giai-96” | “giai-202”

GDTITagURIBody ::= GDTIEncName “:” NumericComponent “.”
GDTIURIBody

GDTIEncName ::= “gdti-96” | “gdti-113”

GSRNTagURIBody ::= GSRNEncName “:” NumericComponent “.”
GSRNURIBody

GSRNEncName ::= “gsrn-96”

GIDTagURIBody ::= GIDEncName “:” GIDURIBody

GIDEncName ::= “gid-96”

DODTagURIBody ::= DODEncName “:” NumericComponent “.”
DODURIBody

DODEncName ::= “dod-96”

RawURI ::= “urn:epc:raw:” RawURIControlBody

RawURIControlBody ::= ( ControlField+ “:”)? RawURIBody

RawURIBody ::=  DecimalRawURIBody | HexRawURIBody |
AFIRawURIBody

DecimalRawURIBody ::= NonZeroComponent “.” NumericComponent

HexRawURIBody ::= NonZeroComponent “.x” HexComponent
```



```
AFIRawURIBody ::= NonZeroComponent ".x" HexComponent ".x"
HexComponent
ControlField ::= "[" ControlName "=" ControlValue "]"
ControlName ::= "att" | "umi" | "xpc"
ControlValue ::= BinaryControlValue | HexControlValue
BinaryControlValue ::= "0" | "1"
HexControlValue ::= "x" HexComponent
```

13 URIs for EPC Patterns

Certain software applications need to specify rules for filtering lists of tags according to various criteria. This specification provides an EPC Tag Pattern URI for this purpose. An EPC Tag Pattern URI does not represent a single tag encoding, but rather refers to a set of tag encodings. A typical pattern looks like this:

`urn:epc:pat:sgtin-96:3.0652642.[102400-204700].*`

This pattern refers to any tag containing a 96-bit SGTIN EPC Binary Encoding, whose Filter field is 3, whose GS1 Company Prefix is 0652642, whose Item Reference is in the range 102400 ≤ *itemReference* ≤ 204700, and whose Serial Number may be anything at all.

In general, there is an EPC Tag Pattern URI scheme corresponding to each EPC Binary Encoding scheme, whose syntax is essentially identical except that ranges or the star (`*`) character may be used in each field.

For the SGTIN, SSCC, SGLN, GRAI, GIAI, GSRN and GDTI patterns, the pattern syntax slightly restricts how wildcards and ranges may be combined. Only two possibilities are permitted for the *`CompanyPrefix`* field. One, it may be a star (`*`), in which case the following field (*`ItemReference`*, *`SerialReference`*, *`LocationReference`*, *`AssetType`*,*`IndividualAssetReference`*, *`ServiceReference`* or *`DocumentType`*) must also be a star. Two, it may be a specific company prefix, in which case the following field may be a number, a range, or a star. A range may not be specified for the *`CompanyPrefix`*.

Explanation (non-normative): Because the company prefix is variable length, a range may not be specified, as the range might span different lengths. When a particular company prefix is specified, however, it is possible to match ranges or all values of the following field, because its length is fixed for a given company prefix. The other case that is allowed is when both fields are a star, which works for all tag encodings because the corresponding tag fields (including the Partition field, where present) are simply ignored.

The pattern URI for the DoD Construct is as follows:

`urn:epc:pat:usdod-96:`*`filterPat.CAGECodeOrDODAACPat.serialNumberPat`*

where *`filterPat`* is either a filter value, a range of the form `[`*`lo-hi`*`]`, or a `*` character; *`CAGECodeOrDODAACPat`* is either a CAGE Code/DODAAC or a `*` character; and *serialNumberPat* is either a serial number, a range of the form [*lo-hi*], or a * character.

13.1 Syntax

The syntax of EPC Tag Pattern URIs is defined by the grammar below.

```
PatURI ::= “urn:epc:pat:” PatBody

PatBody ::= GIDPatURIBody | SGTINPatURIBody |
SGTINAlphaPatURIBody | SGLNGRAI96PatURIBody |
SGLNGRAIAlphaPatURIBody | SSCCPatURIBody | GIAI96PatURIBody
| GIAIAlphaPatURIBody | GSRNPatURIBody | GDTIPatURIBody

GIDPatURIBody ::= “gid-96:” 2*(PatComponent “.”)
PatComponent

SGTIN96PatURIBody ::= “sgtin-96:” PatComponent “.”
GS1PatBody “.” PatComponent

SGTINAlphaPatURIBody ::= “sgtin-198:” PatComponent “.”
GS1PatBody “.” GS3A3PatComponent

SGLNGRAI96PatURIBody ::= SGLNGRAI96TagEncName “:”
PatComponent “.” GS1EPatBody “.” PatComponent

SGLNGRAI96TagEncName ::= “sgln-96” | “grai-96”

SGLNGRAIAlphaPatURIBody ::= SGLNGRAIAlphaTagEncName “:”
PatComponent “.” GS1EPatBody “.” GS3A3PatComponent

SGLNGRAIAlphaTagEncName ::= “sgln-195” | “grai-170”

SSCCPatURIBody ::= “sscc-96:” PatComponent “.” GS1PatBody

GIAI96PatURIBody ::= “giai-96:” PatComponent “.” GS1PatBody

GIAIAlphaPatURIBody ::= “giai-202:” PatComponent “.”
GS1GS3A3PatBody

GSRNPatURIBody ::= “gsrn-96:” PatComponent “.” GS1PatBody

GDTIPatURIBody ::= GDTI96PatURIBody | GDTI113PatURIBody

GDTI96PatURIBody ::= “gdti-96:” PatComponent “.”
GS1EPatBody “.” PatComponent

GDTI113PatURIBody ::= “gdti-113:” PatComponent “.”
GS1EPatBody “.” PaddedNumericOrStarComponent

PaddedNumericOrStarComponent ::= PaddedNumericComponent
                               | StarComponent

GS1PatBody ::= “*.*” | ( PaddedNumericComponent “.”
PaddedPatComponent )

GS1EPatBody ::= “*.*” | ( PaddedNumericComponent “.”
PaddedOrEmptyPatComponent )
```



```
GS1GS3A3PatBody ::= "*.*" | ( PaddedNumericComponent "."
GS3A3PatComponent )

PatComponent ::= NumericComponent
               | StarComponent
               | RangeComponent

PaddedPatComponent ::= PaddedNumericComponent
                     | StarComponent
                     | RangeComponent

PaddedOrEmptyPatComponent ::= PaddedNumericComponentOrEmpty
                            | StarComponent
                            | RangeComponent

GS3A3PatComponent ::= GS3A3Component | StarComponent

StarComponent ::= "*"

RangeComponent ::= "[" NumericComponent "-"
                       NumericComponent "]"
```

For a `RangeComponent` to be legal, the numeric value of the first `NumericComponent` must be less than or equal to the numeric value of the second `NumericComponent`.

13.2 Semantics

The meaning of an EPC Tag Pattern URI (`urn:epc:pat:`) is formally defined as denoting a set of EPC Tag URIs.

The set of EPCs denoted by a specific EPC Tag Pattern URI is defined by the following decision procedure, which says whether a given EPC Tag URI belongs to the set denoted by the EPC Tag Pattern URI.

Let `urn:epc:pat:`*`EncName`*`:P1.P2...P`*`n`* be an EPC Tag Pattern URI. Let `urn:epc:tag:`*`EncName`*`:C1.C2...C`*`n`* be an EPC Tag URI, where the *`EncName`* field of both URIs is the same. The number of components (*`n`*) depends on the value of *`EncName`*.

First, any EPC Tag URI component `C`*`i`* is said to *match* the corresponding EPC Tag Pattern URI component `P`*`i`* if:

- `P`*`i`* is a `NumericComponent`, and `C`*`i`* is equal to `P`*`i`*; or
- `P`*`i`* is a `PaddedNumericComponent`, and `C`*`i`* is equal to `P`*`i`* both in numeric value as well as in length; or
- `P`*`i`* is a `GS3A3Component`, and `C`*`i`* is equal to `P`*`i`*, character for character; or
- `P`*`i`* is a `CAGECodeOrDODAAC`, and `C`*`i`* is equal to `P`*`i`*; or
- `P`*`i`* is a `RangeComponent` `[`*`lo`*`-`*`hi`*`]`, and $lo \le Ci \le hi$; or

- `Pi` is a `StarComponent` (and `Ci` is anything at all)

Then the EPC Tag URI is a member of the set denoted by the EPC Pattern URI if and only if `Ci` matches `Pi` for all $1 \le i \le n$.

14 EPC Binary Encoding

This section specifies how EPC Tag URIs are encoded into binary strings, and conversely how a binary string is decoded into an EPC Tag URI (if possible). The binary strings defined by the encoding and decoding procedures herein are suitable for use in the EPC memory bank of a Gen 2 tag, as specified in Section 15.

The complete procedure for encoding an EPC Tag URI into the binary contents of the EPC memory bank of a Gen 2 tag is specified in Section 15.1.1. The procedure in Section 15.1.1 uses the procedure defined below in Section 14.3 to do the bulk of the work. Conversely, the complete procedure for decoding the binary contents of the EPC memory bank of a Gen 2 tag into an EPC Tag URI (or EPC Raw URI, if necessary) is specified in Section 15.2.2. The procedure in Section 15.2.2 uses the procedure defined below in Section 14.4 to do the bulk of the work.

14.1 Overview of Binary Encoding

The general structure of an EPC Binary Encoding as used on a tag is as a string of bits (i.e., a binary representation), consisting of a fixed length header followed by a series of fields whose overall length, structure, and function are determined by the header value. The assigned header values are specified in Section 14.2.

The procedures for converting between the EPC Tag URI and the binary encoding are specified in Section 14.3 (encoding URI to binary) and Section 14.4 (decoding binary to URI). Both the encoding and decoding procedures are driven by coding tables specified in Section 14.5. Each coding table specifies, for a given header value, the structure of the fields following the header.

To convert an EPC Tag URI to the EPC Binary Encoding, follow the procedure specified in Section 14.3, which is summarized as follows. First, the appropriate coding table is selected from among the tables specified in Section 14.5. The correct coding table is the one whose “URI Template” entry matches the given EPC Tag URI. Each column in the coding table corresponds to a bit field within the final binary encoding. Within each column, a “Coding Method” is specified that says how to calculate the corresponding bits of the binary encoding, given some portion of the URI as input. The encoding details for each “Coding Method” are given in subsections of Section 14.3.

To convert an EPC Binary Encoding into an EPC Tag URI, follow the procedure specified in Section 14.4, which is summarized as follows. First, the most significant eight bits are looked up in the table of EPC binary headers (Table 16 in Section 14.2). This identifies the EPC coding scheme, which in turn selects a coding table from among those specified in Section 14.5. Each column in the coding table corresponds to a bit field in the input binary encoding. Within each column, a “Coding Method” is specified that says how to calculate a corresponding portion of the output URI, given that bit field as input. The decoding details for each “Coding Method” are given in subsections of Section 14.4.

14.2 EPC Binary Headers

The general structure of an EPC Binary Encoding as used on a tag is as a string of bits (i.e., a binary representation), consisting of a fixed length, 8 bit, header followed by a series of fields whose overall length, structure, and function are determined by the header value. For future expansion purpose, a header value of 11111111 is defined, to indicate that longer header beyond 8 bits is used; this provides for future expansion so that more than 256 header values may be accommodated by using longer headers. Therefore, the present specification provides for up to 255 8-bit headers, plus a currently undetermined number of longer headers.

Back-compatibility note (non-normative) In a prior version of the Tag Data Standard, the header was of variable length, using a tiered approach in which a zero value in each tier indicated that the header was drawn from the next longer tier. For the encodings defined in the earlier specification, headers were either 2 bits or 8 bits. Given that a zero value is reserved to indicate a header in the next longer tier, the 2-bit header had 3 possible values (01, 10, and 11, not 00), and the 8-bit header had 63 possible values (recognizing that the first 2 bits must be 00 and 00000000 is reserved to allow headers that are longer than 8 bits). The 2-bit headers were only used in conjunction with certain 64-bit EPC Binary Encodings.

In this version of the Tag Data Standard, the tiered header approach has been abandoned. Also, all 64-bit encodings (including all encodings that used 2-bit headers) have been deprecated, and should not be used in new applications. To facilitate an orderly transition, the portions of header space formerly occupied by 64-bit encodings are reserved in this version of the Tag Data Standard, with the intention that they be reclaimed after a “sunset date” has passed. After the “sunset date,” tags containing 64-bit EPCs with 2-bit headers and tags with 64-bit headers starting with 00001 will no longer be properly interpreted.

Sixteen encoding schemes have been defined in this version of the EPC Tag Data Standard, as shown in Table 1 below. The table also indicates header values that are currently unassigned, as well as header values that have been reserved to allow for an orderly “sunset” of 64-bit encodings defined in prior versions of the EPC Tag Data Standard. These will not be available for assignment until after the “sunset date” has passed. The “sunset date” is July 1, 2009, as stated by EPCglobal on July 1, 2006.

Header Value (binary)	Header Value (hexadecimal)	Encoding Length (bits)	Coding Scheme
0000 0000	00	NA	Unprogrammed Tag

Header Value (binary)	Header Value (hexadecimal)	Encoding Length (bits)	Coding Scheme
0000 0001	01	NA	Reserved for Future Use
0000 001x	02,03	NA	Reserved for Future Use
0000 01xx	04,05	NA	Reserved for Future Use
	06,07	NA	Reserved for Future Use
0000 1000	08	64	Reserved until 64bit Sunset <SSCC-64>
0000 1001	09	64	Reserved until 64bit Sunset <SGLN-64>
0000 1010	0A	64	Reserved until 64bit Sunset <GRAI-64>
0000 1011	0B	64	Reserved until 64bit Sunset <GIAI-64>
0000 1100 to 0000 1111	0C to 0F		Reserved until 64 bit Sunset Due to 64 bit encoding rule in Gen 1
0001 0000 to 0010 1011	10 to 2B	NA NA	Reserved for Future Use
0010 1100	2C	96	GDTI-96
0010 1101	2D	96	GSRN-96
0010 1110	2E	NA	Reserved for Future Use
0010 1111	2F	96	DoD-96
0011 0000	30	96	SGTIN-96
0011 0001	31	96	SSCC-96
0011 0010	32	96	SGLN-96
0011 0011	33	96	GRAI-96
0011 0100	34	96	GIAI-96
0011 0101	35	96	GID-96
0011 0110	36	198	SGTIN-198
0011 0111	37	170	GRAI-170
0011 1000	38	202	GIAI-202
0011 1001	39	195	SGLN-195
0011 1010	3A	113	GDTI-113

Header Value (binary)	Header Value (hexadecimal)	Encoding Length (bits)	Coding Scheme
0011 1011 to 0011 1111	3B to 3F	NA	Reserved for future Header values
0100 0000 to 0111 1111	40 to 7F		Reserved until 64 bit Sunset
1000 0000 to 1011 1111	80 to BF	64	Reserved until 64 bit Sunset <SGTIN-64> (64 header values)
1100 0000 to 1100 1101	C0 to CD		Reserved until 64 bit Sunset
1100 1110	CE	64	Reserved until 64 bit Sunset <DoD-64>
1100 1111 to 1111 1110	CF to FE		Reserved until 64 bit Sunset Following 64 bit Sunset, E2 remains reserved to avoid confusion with the first eight bits of TID memory (Section 16).
1111 1111	FF	NA	Reserved for future headers longer than 8 bits

Table 16. EPC Binary Header Values

14.3 Encoding Procedure

The following procedure encodes an EPC Tag URI into a bit string containing the encoded EPC and (for EPC schemes that have a filter value) the filter value. This bit string is suitable for storing in the EPC memory bank of a Gen 2 Tag beginning at bit 20_h. See Section 15.1.1 for the complete procedure for encoding the entire EPC memory bank, including control information that resides outside of the encoded EPC. (The procedure in Section 15.1.1 uses the procedure below as a subroutine.)

Given:

- An EPC Tag URI of the form `urn:epc:tag:`*`scheme`*`:`*`remainder`*

Yields:

- A bit string containing the EPC binary encoding of the specified EPC Tag URI, containing the encoded EPC together with the filter value (if applicable); OR

- An exception indicating that the EPC Tag URI could not be encoded.

Procedure:

1. Use the `scheme` to identify the coding table for this URI scheme. If no such scheme exists, stop: this URI is not syntactically legal.
2. Confirm that the URI syntactically matches the URI template associated with the coding table. If not, stop: this URI is not syntactically legal.
3. Read the coding table left-to-right, and construct the encoding specified in each column to obtain a *b*-bit string, where *b* is specified in the "Coding Segment Bit Count" row of the table. The method for encoding each column depends on the "Coding Method" row of the table. If the "Coding Method" row specifies a specific bit string, use that bit string for that column. Otherwise, consult the following sections that specify the encoding methods. If the encoding of any segment fails, stop: this URI cannot be encoded.
4. Concatenate the bit strings from Step 3 to form a single *B*-bit string, where *B* is the overall binary length specified by the scheme. The position of each segment within the concatenated bit string is as specified in the "Bit Position" row of the coding table. Section 15.1.1 specifies the procedure that uses the result of this step for encoding the EPC memory bank of a Gen 2 tag.

The following sections specify the procedures to be used in Step 3.

14.3.1 "Integer" Encoding Method

The Integer encoding method is used for a segment that appears as a decimal integer in the URI, and as a binary integer in the binary encoding.

Input: The input to the encoding method is the URI portion indicated in the "URI portion" row of the encoding table, a character string with no dot ("`.`") characters.

Validity Test: The input character string must satisfy the following:

- It must match the grammar for `NumericComponent` as specified in Section 5.
- The value of the string when considered as a decimal integer must be less than 2^b, where *b* is the value specified in the "Coding Segmen Bit Count" row of the encoding table.

If any of the above tests fails, the encoding of the URI fails.

Output: The encoding of this segment is a *b*-bit integer, where *b* is the value specified in the "Coding Segment Bit Count" row of the encoding table, whose value is the value of the input character string considered as a decimal integer.

14.3.2 "String" Encoding Method

The String encoding method is used for a segment that appears as an alphanumeric string in the URI, and as an ISO 646 (ASCII) encoded bit string in the binary encoding.

Input: The input to the encoding method is the URI portion indicated in the "URI portion" row of the encoding table, a character string with no dot ("`.`") characters.

Validity Test: The input character string must satisfy the following:

- It must match the grammar for `GS3A3Component` as specified in Section 5.
- For each portion of the string that matches the `Escape` production of the grammar specified in Section 5 (that is, a 3-character sequence consisting of a `%` character followed by two hexadecimal digits), the two hexadecimal characters following the `%` character must map to one of the 82 allowed characters specified in Table 46 (Appendix A).
- The number of characters must be less than $b/7$, where b is the value specified in the "Coding Segment Bit Count" row of the coding table.

If any of the above tests fails, the encoding of the URI fails.

Output: Consider the input to be a string of zero or more characters $s_1s_2\ldots s_N$, where each character s_i is either a single character or a 3-character sequence matching the `Escape` production of the grammar (that is, a 3-character sequence consisting of a `%` character followed by two hexadecimal digits). Translate each character to a 7-bit string. For a single character, the corresponding 7-bit string is specified in Table 46 (Appendix A). For an `Escape` sequence, the 7-bit string is the value of the two hexadecimal characters considered as a 7-bit integer. Concatenating those 7-bit strings in the order corresponding to the input, then pad with zero bits as necessary to total b bits, where b is the value specified in the "Coding Segment Bit Count" row of the coding table. (The number of padding bits will be $b - 7N$.) The resulting b-bit string is the output.

14.3.3 "Partition Table" Encoding Method

The Partition Table encoding method is used for a segment that appears in the URI as a pair of variable-length numeric fields separated by a dot ("`.`") character, and in the binary encoding as a 3-bit "partition" field followed by two variable length binary integers. The number of characters in the two URI fields always totals to a constant number of characters, and the number of bits in the binary encoding likewise totals to a constant number of bits.

The Partition Table encoding method makes use of a "partition table." The specific partition table to use is specified in the coding table for a given EPC scheme.

Input: The input to the encoding method is the URI portion indicated in the "URI portion" row of the encoding table. This consists of two strings of digits separated by a dot ("`.`") character. For the purpose of this encoding procedure, the digit strings to the left and right of the dot are denoted C and D, respectively.

Validity Test: The input must satisfy the following:

- C must match the grammar for `PaddedNumericComponent` as specified in Section 5.

- *D* must match the grammar for `PaddedNumericComponentOrEmpty` as specified in Section 5.
- The number of digits in *C* must match one of the values specified in the “GS1 Company Prefix Digits (L)” column of the partition table. The corresponding row is called the “matching partition table row” in the remainder of the encoding procedure.
- The number of digits in *D* must match the corresponding value specified in the “Other Field Digits” column of the matching partition table row. Note that if the “Other Field Digits” column specifies zero, then *D* must be the empty string, implying the overall input segment ends with a “dot” character.

Output: Construct the output bit string by concatenating the following three components:

- The value *P* specified in the “partition value” column of the matching partition table row, as a 3-bit binary integer.
- The value of *C* considered as a decimal integer, converted to an *M*-bit binary integer, where *M* is the number of bits specified in the “GS1 Company Prefix bits” column of the matching partition table row.
- The value of *D* considered as a decimal integer, converted to an *N*-bit binary integer, where *N* is the number of bits specified in the “other field bits” column of the matching partition table row. If *D* is the empty string, the value of the *N*-bit integer is zero.

The resulting bit string is $(3 + M + N)$ bits in length, which always equals the “Coding Segment Bit Count” for this segment as indicated in the coding table.

14.3.4 “Unpadded Partition Table” Encoding Method

The Unpadded Partition Table encoding method is used for a segment that appears in the URI as a pair of variable-length numeric fields separated by a dot (“`.`”) character, and in the binary encoding as a 3-bit “partition” field followed by two variable length binary integers. The number of characters in the two URI fields is always less than or equal to a known limit, and the number of bits in the binary encoding is always a constant number of bits.

The Unpadded Partition Table encoding method makes use of a “partition table.” The specific partition table to use is specified in the coding table for a given EPC scheme.

Input: The input to the encoding method is the URI portion indicated in the “URI portion” row of the encoding table. This consists of two strings of digits separated by a dot (“`.`”) character. For the purpose of this encoding procedure, the digit strings to the left and right of the dot are denoted *C* and *D*, respectively.

Validity Test: The input must satisfy the following:

- *C* must match the grammar for `PaddedNumericComponent` as specified in Section 5.
- *D* must match the grammar for `NumericComponent` as specified in Section 5.

- The number of digits in *C* must match one of the values specified in the "GS1 Company Prefix Digits (L)" column of the partition table. The corresponding row is called the "matching partition table row" in the remainder of the encoding procedure.
- The value of *D*, considered as a decimal integer, must be less than 2^N, where *N* is the number of bits specified in the "other field bits" column of the matching partition table row.

Output: Construct the output bit string by concatenating the following three components:

- The value *P* specified in the "partition value" column of the matching partition table row, as a 3-bit binary integer.
- The value of *C* considered as a decimal integer, converted to an *M*-bit binary integer, where *M* is the number of bits specified in the "GS1 Company Prefix bits" column of the matching partition table row.
- The value of *D* considered as a decimal integer, converted to an *N*-bit binary integer, where *N* is the number of bits specified in the "other field bits" column of the matching partition table row. If *D* is the empty string, the value of the *N*-bit integer is zero.

The resulting bit string is (3 + *M* + *N*) bits in length, which always equals the "Coding Segment Bit Count" for this segment as indicated in the coding table.

14.3.5 "String Partition Table" Encoding Method

The String Partition Table encoding method is used for a segment that appears in the URI as a variable-length numeric field and a variable-length string field separated by a dot ("`.`") character, and in the binary encoding as a 3-bit "partition" field followed by a variable length binary integer and a variable length binary-encoded character string. The number of characters in the two URI fields is always less than or equal to a known limit (counting a 3-character escape sequence as a single character), and the number of bits in the binary encoding is padded if necessary to a constant number of bits.

The Partition Table encoding method makes use of a "partition table." The specific partition table to use is specified in the coding table for a given EPC scheme.

Input: The input to the encoding method is the URI portion indicated in the "URI portion" row of the encoding table. This consists of two strings separated by a dot ("`.`") character. For the purpose of this encoding procedure, the strings to the left and right of the dot are denoted *C* and *D*, respectively.

Validity Test: The input must satisfy the following:

- *C* must match the grammar for `PaddedNumericComponent` as specified in Section 5.
- *D* must match the grammar for `GS3A3Component` as specified in Section 5.
- The number of digits in *C* must match one of the values specified in the "GS1 Company Prefix Digits (L)" column of the partition table. The corresponding row is called the "matching partition table row" in the remainder of the encoding procedure.

- The number of characters in *D* must be less than or equal to the corresponding value specified in the "Other Field Maximum Characters" column of the matching partition table row. For the purposes of this rule, an escape triplet (`%nn`) is counted as one character.
- For each portion of *D* that matches the `Escape` production of the grammar specified in Section 5 (that is, a 3-character sequence consisting of a `%` character followed by two hexadecimal digits), the two hexadecimal characters following the `%` character must map to one of the 82 allowed characters specified in Table 46 (Appendix A).

Output: Construct the output bit string by concatenating the following three components:

- The value *P* specified in the "partition value" column of the matching partition table row, as a 3-bit binary integer.
- The value of *C* considered as a decimal integer, converted to an *M*-bit binary integer, where *M* is the number of bits specified in the "GS1 Company Prefix bits" column of the matching partition table row.
- The value of *D* converted to an *N*-bit binary string, where *N* is the number of bits specified in the "other field bits" column of the matching partition table row. This *N*-bit binary string is constructed as follows. Consider *D* to be a string of zero or more characters $s_1 s_2 \ldots s_N$, where each character s_i is either a single character or a 3-character sequence matching the `Escape` production of the grammar (that is, a 3-character sequence consisting of a `%` character followed by two hexadecimal digits). Translate each character to a 7-bit string. For a single character, the corresponding 7-bit string is specified in Table 46 (Appendix A). For an `Escape` sequence, the 7-bit string is the value of the two hexadecimal characters considered as a 7-bit integer. Concatenate those 7-bit strings in the order corresponding to the input, then pad with zero bits as necessary to total *N* bits.

The resulting bit string is $(3 + M + N)$ bits in length, which always equals the "Coding Segment Bit Count" for this segment as indicated in the coding table.

14.3.6 "Numeric String" Encoding Method

The Numeric String encoding method is used for a segment that appears as a numeric string in the URI, possibly including leading zeros. The leading zeros are preserved in the binary encoding by prepending a "1" digit to the numeric string before encoding.

Input: The input to the encoding method is the URI portion indicated in the "URI portion" row of the encoding table, a character string with no dot ("`.`") characters.

Validity Test: The input character string must satisfy the following:

- It must match the grammar for `PaddedNumericComponent` as specified in Section 5.
- The number of digits in the string, D, must be such that $2 \times 10^{D} < 2^{b}$, where *b* is the value specified in the "Coding Segment Bit Count" row of the encoding table. (For

the GDTI-113 scheme, b = 58 and therefore the number of digits D must be less than or equal to 17. GDTI-113 is the only scheme that uses this encoding method.)

If any of the above tests fails, the encoding of the URI fails.

Output: Construct the output bit string as follows:

- Prepend the character "`1`" to the left of the input character string.
- Convert the resulting string to a b-bit integer, where b is the value specified in the "bit count" row of the encoding table, whose value is the value of the input character string considered as a decimal integer.

14.4 Decoding Procedure

This procedure decodes a bit string as found beginning at bit 20_h in the EPC memory bank of a Gen 2 Tag into an EPC Tag URI. This procedure only decodes the EPC and filter value (if applicable). Section 15.2.2 gives the complete procedure for decoding the entire contents of the EPC memory bank, including control information that is stored outside of the encoded EPC. The procedure in Section 15.2.2 should be used by most applications. (The procedure in Section 15.2.2 uses the procedure below as a subroutine.)

Given:

- A bit string consisting of N bits $b_{N-1}b_{N-2}\ldots b_0$

Yields:

- An EPC Tag URI beginning with `urn:epc:tag:`, which does not contain control information fields (other than the filter value if the EPC scheme includes a filter value); OR
- An exception indicating that the bit string cannot be decoded into an EPC Tag URI.

Procedure:

1. Extract the most significant eight bits, the EPC header: $b_{\mathrm{N}-1}b_{\mathrm{N}-2}\ldots b_{\mathrm{N}-8}$. Referring to Table 16 in Section 14.2, use the header to identify the coding table for this binary encoding and the encoding bit length B. If no coding table exists for this header, stop: this binary encoding cannot be decoded.
2. Confirm that the total number of bits N is greater than or equal to the total number of bits B specified for this header in Table 16. If not, stop: this binary encoding cannot be decoded.
3. If necessary, truncate the least significant bits of the input to match the number of bits specified in Table 16. That is, if Table 16 specifies B bits, retain bits $b_{\mathrm{N}-1}b_{\mathrm{N}-2}\ldots b_{\mathrm{N}-\mathrm{B}}$. For the remainder of this procedure, consider the remaining bits to be numbered $b_{B-1}b_{B-2}\ldots b_0$. (The purpose of this step is to remove any trailing zero padding bits that may have been read due to word-oriented data transfer.)
4. Separate the bits of the binary encoding into segments according to the "bit position" row of the coding table. For each segment, decode the bits to obtain a character string that will be used as a portion of the final URI. The method for decoding each column

depends on the "coding method" row of the table. If the "coding method" row specifies a specific bit string, the corresponding bits of the input must match those bits exactly; if not, stop: this binary encoding cannot be decoded. Otherwise, consult the following sections that specify the decoding methods. If the decoding of any segment fails, stop: this binary encoding cannot be decoded.

5. Concatenate the following strings to obtain the final URI: the string `urn:epc:tag:`, the scheme name as specified in the coding table, a colon ("`:`") character, and the strings obtained in Step 3, inserting a dot ("`.`") character between adjacent strings.

The following sections specify the procedures to be used in Step 3.

14.4.1 "Integer" Decoding Method

The Integer decoding method is used for a segment that appears as a decimal integer in the URI, and as a binary integer in the binary encoding.

Input: The input to the decoding method is the bit string identified in the "bit position" row of the coding table.

Validity Test: There are no validity tests for this decoding method.

Output: The decoding of this segment is a decimal numeral whose value is the value of the input considered as an unsigned binary integer. The output shall not begin with a zero character if it is two or more digits in length.

14.4.2 "String" Decoding Method

The String decoding method is used for a segment that appears as a alphanumeric string in the URI, and as an ISO 646 (ASCII) encoded bit string in the binary encoding.

Input: The input to the decoding method is the bit string identified in the "bit position" row of the coding table. This length of this bit string is always a multiple of seven.

Validity Test: The input bit string must satisfy the following:

- Each 7-bit segment must have a value corresponding to a character specified in Table 46 (Appendix A), or be all zeros.
- All 7-bit segments following an all-zero segment must also be all zeros.
- The first 7-bit segment must not be all zeros. (In other words, the string must contain at least one character.)

If any of the above tests fails, the decoding of the segment fails.

Output: Translate each 7-bit segment, up to but not including the first all-zero segment (if any), into a single character or 3-charcter escape triplet by looking up the 7-bit segment in Table 46 (Appendix A) and using the value found in the "URI Form" column. Concatenate the characters and/or 3-character triplets in the order corresponding to the input bit string. The resulting character string is the output. This character string matches the `GS3A3` production of the grammar in Section 5.

14.4.3 "Partition Table" Decoding Method

The Partition Table decoding method is used for a segment that appears in the URI as a pair of variable-length numeric fields separated by a dot ("`.`") character, and in the binary encoding as a 3-bit "partition" field followed by two variable length binary integers. The number of characters in the two URI fields always totals to a constant number of characters, and the number of bits in the binary encoding likewise totals to a constant number of bits.

The Partition Table decoding method makes use of a "partition table." The specific partition table to use is specified in the coding table for a given EPC scheme.

Input: The input to the decoding method is the bit string identified in the "bit position" row of the coding table. Logically, this bit string is divided into three substrings, consisting of a 3-bit "partition" value, followed by two substrings of variable length.

Validity Test: The input must satisfy the following:

- The three most significant bits of the input bit string, considered as a binary integer, must match one of the values specified in the "partition value" column of the partition table. The corresponding row is called the "matching partition table row" in the remainder of the decoding procedure.
- Extract the M next most significant bits of the input bit string following the three partition bits, where M is the value specified in the "Compay Prefix Bits" column of the matching partition table row. Consider these M bits to be an unsigned binary integer, C. The value of C must be less than 10^L, where L is the value specified in the "GS1 Company Prefix Digits (L)" column of the matching partition table row.
- There are N bits remaining in the input bit string, where N is the value specified in the "Other Field Bits" column of the matching partition table row. Consider these N bits to be an unsigned binary integer, D. The value of D must be less than 10^K, where K is the value specified in the "Other Field Digits (K)" column of the matching partition table row. Note that if $K = 0$, then the value of D must be zero.

Output: Construct the output character string by concatenating the following three components:

- The value C converted to a decimal numeral, padding on the left with zero ("`0`") characters to make L digits in total.
- A dot ("`.`") character.
- The value D converted to a decimal numeral, padding on the left with zero ("`0`") characters to make K digits in total. If $K = 0$, append no characters to the dot above (in this case, the final URI string will have two adjacent dot characters when this segment is combined with the following segment).

14.4.4 "Unpadded Partition Table" Decoding Method

The Unpadded Partition Table decoding method is used for a segment that appears in the URI as a pair of variable-length numeric fields separated by a dot ("`.`") character, and in the binary encoding as a 3-bit "partition" field followed by two variable length binary integers. The number of characters in the two URI fields is always less than or equal to a known limit, and the number of bits in the binary encoding is always a constant number of bits.

The Unpadded Partition Table decoding method makes use of a "partition table." The specific partition table to use is specified in the coding table for a given EPC scheme.

Input: The input to the decoding method is the bit string identified in the "bit position" row of the coding table. Logically, this bit string is divided into three substrings, consisting of a 3-bit "partition" value, followed by two substrings of variable length.

Validity Test: The input must satisfy the following:

- The three most significant bits of the input bit string, considered as a binary integer, must match one of the values specified in the "partition value" column of the partition table. The corresponding row is called the "matching partition table row" in the remainder of the decoding procedure.
- Extract the M next most significant bits of the input bit string following the three partition bits, where M is the value specified in the "Compay Prefix Bits" column of the matching partition table row. Consider these M bits to be an unsigned binary integer, C. The value of C must be less than 10^L, where L is the value specified in the "GS1 Company Prefix Digits (L)" column of the matching partition table row.
- There are N bits remaining in the input bit string, where N is the value specified in the "Other Field Bits" column of the matching partition table row. Consider these N bits to be an unsigned binary integer, D. The value of D must be less than 10^K, where K is the value specified in the "Other Field Max Digits (K)" column of the matching partition table row.

Output: Construct the output character string by concatenating the following three components:

- The value C converted to a decimal numeral, padding on the left with zero ("`0`") characters to make L digits in total.
- A dot ("`.`") character.
- The value D converted to a decimal numeral, with no leading zeros (except that if $D = 0$ it is converted to a single zero digit).

14.4.5 "String Partition Table" Decoding Method

The String Partition Table decoding method is used for a segment that appears in the URI as a variable-length numeric field and a variable-length string field separated by a dot ("`.`") character, and in the binary encoding as a 3-bit "partition" field followed by a variable length binary integer and a variable length binary-encoded character string. The number of characters in the two URI fields is always less than or equal to a known limit (counting a 3-character escape sequence as a single character), and the number of bits in the binary encoding is padded if necessary to a constant number of bits.

The Partition Table decoding method makes use of a “partition table.” The specific partition table to use is specified in the coding table for a given EPC scheme.

Input: The input to the decoding method is the bit string identified in the “bit position” row of the coding table. Logically, this bit string is divided into three substrings, consisting of a 3-bit “partition” value, followed by two substrings of variable length.

Validity Test: The input must satisfy the following:

- The three most significant bits of the input bit string, considered as a binary integer, must match one of the values specified in the “partition value” column of the partition table. The corresponding row is called the “matching partition table row” in the remainder of the decoding procedure.
- Extract the M next most significant bits of the input bit string following the three partition bits, where M is the value specified in the “Compay Prefix Bits” column of the matching partition table row. Consider these M bits to be an unsigned binary integer, C. The value of C must be less than 10^L, where L is the value specified in the “GS1 Company Prefix Digits (L)” column of the matching partition table row.
- There are N bits remaining in the input bit string, where N is the value specified in the “Other Field Bits” column of the matching partition table row. These bits must consist of one or more non-zero 7-bit segments followed by zero or more all-zero bits.
- The number of non-zero 7-bit segments that precede the all-zero bits (if any) must be less or equal to than K, where K is the value specified in the “Maximum Characters” column of the matching partition table row.
- Each of the non-zero 7-bit segments must have a value corresponding to a character specified in Table 46 (Appendix A).

Output: Construct the output character string by concatenating the following three components:

- The value C converted to a decimal numeral, padding on the left with zero (“`0`”) characters to make L digits in total.
- A dot (“`.`”) character.
- A character string determined as follows. Translate each non-zero 7-bit segment as determined by the validity test into a single character or 3-character escape triplet by looking up the 7-bit segment in Table 46 (Appendix A) and using the value found in the “URI Form” column. Concatenate the characters and/or 3-character triplet in the order corresponding to the input bit string.

14.4.6 “Numeric String” Decoding Method

The Numeric String decoding method is used for a segment that appears as a numeric string in the URI, possibly including leading zeros. The leading zeros are preserved in the binary encoding by prepending a “1” digit to the numeric string before encoding.

Input: The input to the decoding method is the bit string identified in the “bit position” row of the coding table.

Validity Test: The input must be such that the decoding procedure below does not fail.

Output: Construct the output string as follows.

- Convert the input bit string to a decimal numeral without leading zeros whose value is the value of the input considered as an unsigned binary integer.
- If the numeral from the previous step does not begin with a “`1`” character, stop: the input is invalid.
- If the numeral from the previous step consists only of one character, stop: the input is invalid (because this would correspond to an empty numeric string).
- Delete the leading “`1`” character from the numeral.
- The resulting string is the output.

14.5 EPC Binary Coding Tables

This section specifies coding tables for use with the encoding procedure of Section 14.3 and the decoding procedure of Section 14.3.4.

The “Bit Position” row of each coding table illustrates the relative bit positions of segments within each binary encoding. In the “Bit Position” row, the highest subscript indicates the most significant bit, and subscript 0 indicates the least significant bit. Note that this is opposite to the way RFID tag memory bank bit addresses are normally indicated, where address 0 is the most significant bit.

14.5.1 Serialized Global Trade Identification Number (SGTIN)

Two coding schemes for the SGTIN are specified, a 96-bit encoding (SGTIN-96) and a 198-bit encoding (SGTIN-198). The SGTIN-198 encoding allows for the full range of serial numbers up to 20 alphanumeric characters as specified in [GS1GS10.0]. The SGTIN-96 encoding allows for numeric-only serial numbers, without leading zeros, whose value is less than 2^{38} (that is, from 0 through 274,877,906,943, inclusive).

Both SGTIN coding schemes make reference to the following partition table.

Partition Value (*P*)	**GS1 Company Prefix**		**Indicator/Pad Digit and Item Reference**	
	Bits (*M*)	**Digits (*L*)**	**Bits (N)**	**Digits**
0	40	12	4	1
1	37	11	7	2
2	34	10	10	3

Partition Value (*P*)	GS1 Company Prefix		Indicator/Pad Digit and Item Reference	
	Bits (*M*)	Digits (*L*)	Bits (N)	Digits
3	30	9	14	4
4	27	8	17	5
5	24	7	20	6
6	20	6	24	7

Table 17. SGTIN Partition Table

14.5.1.1 SGTIN-96 Coding Table

Scheme	SGTIN-96					
URI Template	`urn:epc:tag:sgtin-96:`*`F.C.I.S`*					
Total Bits	96					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix (*)	Indicator (**) / Item Reference	Serial
Logical Segment Bit Count	8	3	3	20-40	24-4	38
Coding Segment	EPC Header	Filter	GTIN			Serial
URI portion		*`F`*	*`C.I`*			*`S`*
Coding Segment Bit Count	8	3	47			38
Bit Position	$b_{95}b_{94}\ldots b_{88}$	$b_{87}b_{86}b_{85}$	$b_{84}b_{83}\ldots b_{38}$			$b_{37}b_{36}\ldots b_{0}$
Coding Method	00110000	Integer	Partition Table 17			Integer

Table 18. SGTIN-96 Coding Table

(*) See Section 7.1.2 for the case of an SGTIN derived from a GTIN-8.

(**) Note that in the case of an SGTIN derived from a GTIN-12 or GTIN-13, a zero pad digit takes the place of the Indicator Digit. In all cases, see Section 7.1 for the definition of how the Indicator Digit (or zero pad) and the Item Reference are combined into this segment of the EPC.

14.5.1.2 SGTIN-198 Coding Table

Scheme	SGTIN-198					
URI Template	`urn:epc:tag:sgtin-198:`*`F.C.I.S`*					
Total Bits	198					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix (*)	Indicator (**) / Item Reference	Serial
Logical Segment Bit Count	8	3	3	20-40	24-4	140
Coding Segment	EPC Header	Filter	GTIN			Serial
URI portion		*`F`*	*`C.I`*			*`S`*
Coding Segment Bit Count	8	3	47			140
Bit Position	$b_{197}b_{196}\ldots b_{190}$	$b_{189}b_{188}b_{187}$	$b_{186}b_{185}\ldots b_{140}$			$b_{139}b_{138}\ldots b_{0}$
Coding Method	00110110	Integer	Partition Table 17			String

Table 19. SGTIN-198 Coding Table

(*) See Section 7.1.2 for the case of an SGTIN derived from a GTIN-8.

(**) Note that in the case of an SGTIN derived from a GTIN-12 or GTIN-13, a zero pad digit takes the place of the Indicator Digit. In all cases, see Section 7.1 for the definition of how the Indicator Digit (or zero pad) and the Item Reference are combined into this segment of the EPC.

14.5.2 Serial Shipping Container Code (SSCC)

One coding scheme for the SSCC is specified: the 96-bit encoding SSCC-96. The SSCC-96 encoding allows for the full range of SSCCs as specified in [GS1GS10.0].

The SSCC-96 coding scheme makes reference to the following partition table.

Partition Value (*P*)	**GS1 Company Prefix**		**Extension Digit and Serial Reference**	
	Bits (*M*)	**Digits (*L*)**	**Bits (*N*)**	**Digits**
0	40	12	18	5
1	37	11	21	6
2	34	10	24	7
3	30	9	28	8
4	27	8	31	9
5	24	7	34	10
6	20	6	38	11

Table 20. SSCC Partition Table

14.5.2.1 SSCC-96 Coding Table

Scheme	SSCC-96					
URI Template	`urn:epc:tag:sscc-96:`*`F.C.S`*					
Total Bits	96					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Extension / Serial Reference	(Reserved)
Logical Segment Bit Count	8	3	3	20-40	38-18	24
Coding Segment	EPC Header	Filter	SSCC			(Reserved)
URI portion		*`F`*	*`C.S`*			
Coding Segment Bit Count	8	3	61			24
Bit Position	$b_{95}b_{94}\ldots b_{88}$	$b_{87}b_{86}b_{85}$	$b_{84}b_{83}\ldots b_{24}$			$b_{23}b_{36}\ldots b_{0}$
Coding Method	00110001	Integer	Partition Table 20			00…0 (24 zero bits)

Table 21. SSCC-96 Coding Table

14.5.3 Serialized Global Location Number (SGLN)

Two coding schemes for the SGLN are specified, a 96-bit encoding (SGLN-96) and a 195-bit encoding (SGLN-195). The SGLN-195 encoding allows for the full range of GLN extensions up to 20 alphanumeric characters as specified in [GS1GS10.0]. The SGLN-96 encoding allows for numeric-only GLN extensions, without leading zeros, whose value is less than 2^{41} (that is, from 0 through 2,199,023,255,551, inclusive). Note that an extension value of 0 is reserved to indicate that the SGLN is equivalent to the GLN indicated by the GS1 Company Prefix and location reference; this value is available in both the SGLN-96 and the SGLN-195 encodings.

Both SGLN coding schemes make reference to the following partition table.

Partition Value (*P*)	**GS1 Company Prefix**		**Location Reference**	
	Bits	**Digits**	**Bits**	**Digits**

	(*M*)	**(*L*)**	**(*N*)**	
0	40	12	1	0
1	37	11	4	1
2	34	10	7	2
3	30	9	11	3
4	27	8	14	4
5	24	7	17	5
6	20	6	21	6

Table 22. SGLN Partition Table

14.5.3.1 SGLN-96 Coding Table

Scheme	SGLN-96					
URI Template	`urn:epc:tag:sgln-96:`*`F.C.L.E`*					
Total Bits	96					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Location Reference	Extension
Logical Segment Bit Count	8	3	3	20-40	21-1	41
Coding Segment	EPC Header	Filter	GLN			Extension
URI portion		*`F`*	*`C.L`*			*`E`*
Coding Segment Bit Count	8	3	44			41
Bit Position	$b_{95}b_{94}\ldots b_{88}$	$b_{87}b_{86}b_{85}$	$b_{84}b_{83}\ldots b_{41}$			$b_{40}b_{39}\ldots b_0$
Coding Method	00110010	Integer	Partition Table 22			Integer

Table 23. SGLN-96 Coding Table

14.5.3.2 SGLN-195 Coding Table

Scheme	SGLN-195					
URI Template	`urn:epc:tag:sgln-195:`*`F.C.L.E`*					
Total Bits	195					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Location Reference	Extension
Logical Segment Bit Count	8	3	3	20-40	21-1	140
Coding Segment	EPC Header	Filter	GLN			Extension
URI portion		*`F`*	*`C.L`*			*`E`*
Coding Segment Bit Count	8	3	44			140
Bit Position	$b_{194}b_{193}\ldots b_{187}$	$b_{186}b_{185}b_{184}$	$b_{183}b_{182}\ldots b_{140}$			$b_{139}b_{138}\ldots b_{0}$
Coding Method	00111001	Integer	Partition Table 22			String

Table 24. SGLN-195 Coding Table

14.5.4 Global Returnable Asset Identifier (GRAI)

Two coding schemes for the GRAI are specified, a 96-bit encoding (GRAI-96) and a 170-bit encoding (SGTIN-170). The GRAI-170 encoding allows for the full range of serial numbers up to 16 alphanumeric characters as specified in [GS1GS10.0]. The GRAI-96 encoding allows for numeric-only serial numbers, without leading zeros, whose value is less than 2^{38} (that is, from 0 through 274,877,906,943, inclusive).

Only GRAIs that include the optional serial number may be represented as EPCs. A GRAI without a serial number represents an asset class, rather than a specific instance, and therefore may not be used as an EPC (just as a non-serialized GTIN may not be used as an EPC).

Both GRAI coding schemes make reference to the following partition table.

Partition Value (*P*)	Company Prefix		Asset Type	
	Bits (*M*)	**Digits (*L*)**	**Bits (N)**	**Digits**
0	40	12	4	0
1	37	11	7	1
2	34	10	10	2
3	30	9	14	3
4	27	8	17	4
5	24	7	20	5
6	20	6	24	6

Table 25. GRAI Partition Table

14.5.4.1 GRAI-96 Coding Table

Scheme	GRAI-96					
URI Template	`urn:epc:tag:grai-96:`*`F.C.A.S`*					
Total Bits	96					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Asset Type	Serial
Logical Segment Bit Count	8	3	3	20-40	24-3	38
Coding Segment	EPC Header	Filter	Partition + Company Prefix + Asset Type			Serial
URI portion		*`F`*	*`C.A`*			*`S`*
Coding Segment Bit Count	8	3	47			38
Bit Position	$b_{95}b_{94}\ldots b_{88}$	$b_{87}b_{86}b_{85}$	$b_{84}b_{83}\ldots b_{38}$			$b_{37}b_{36}\ldots b_{0}$
Coding Method	00110011	Integer	Partition Table 25			Integer

Table 26. GRAI-96 Coding Table

14.5.4.2 GRAI-170 Coding Table

Scheme	GRAI-170					
URI Template	`urn:epc:tag:grai-170:`*`F.C.A.S`*					
Total Bits	170					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Asset Type	Serial
Logical Segment Bit Count	8	3	3	20-40	24-3	112
Coding Segment	EPC Header	Filter	Partition + Company Prefix + Asset Type			Serial
URI portion		*`F`*	*`C.A`*			*`S`*
Coding Segment Bit Count	8	3	47			112
Bit Position	$b_{169}b_{168}\ldots b_{162}$	$b_{161}b_{160}b_{159}$	$b_{158}b_{157}\ldots b_{112}$			$b_{111}b_{110}\ldots b_{0}$
Coding Method	00110111	Integer	Partition Table 25			String

Table 27. GRAI-170 Coding Table

14.5.5 Global Individual Asset Identifier (GIAI)

Two coding schemes for the GIAI are specified, a 96-bit encoding (GIAI-96) and a 202-bit encoding (GIAI-202). The GIAI-202 encoding allows for the full range of serial numbers up to 24 alphanumeric characters as specified in [GS1GS10.0]. The GIAI-96 encoding allows for numeric-only serial numbers, without leading zeros, whose value is, up to a limit that varies with the length of the GS1 Company Prefix.

Each GIAI coding schemes make reference to a different partition table, specified alongside the corresponding coding table in the subsections below.

14.5.5.1 GIAI-96 Partition Table and Coding Table

The GIAI-96 coding scheme makes use of the following partition table.

Partition Value (*P*)	Company Prefix		Individual Asset Reference	
	Bits (*M*)	**Digits (*L*)**	**Bits (*N*)**	**Max Digits (K)**
0	40	12	42	13
1	37	11	45	14
2	34	10	48	15
3	30	9	52	16
4	27	8	55	17
5	24	7	58	18
6	20	6	62	19

2652 Table 28. GIAI-96 Partition Table

Scheme	GIAI-96				
URI Template	`urn:epc:tag:giai-96:`*`F.C.A`*				
Total Bits	96				
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Individual Asset Reference
Logical Segment Bit Count	8	3	3	20-40	62–42
Coding Segment	EPC Header	Filter	GIAI		
URI portion		*`F`*	*`C.A`*		
Coding Segment Bit Count	8	3	85		
Bit Position	$b_{95}b_{94}\ldots b_{88}$	$b_{87}b_{86}b_{85}$	$b_{84}b_{83}\ldots b_0$		
Coding Method	00110100	Integer	Unpadded Partition Table 28		

2653 Table 29. GIAI-96 Coding Table

14.5.5.2 GIAI-202 Partition Table and Coding Table

The GIAI-202 coding scheme makes use of the following partition table.

Partition Value (*P*)	**Company Prefix**		**Individual Asset Reference**	
	Bits (*M*)	**Digits (*L*)**	**Bits (*N*)**	**Maximum Characters**
0	40	12	148	18
1	37	11	151	19
2	34	10	154	20
3	30	9	158	21
4	27	8	161	22
5	24	7	164	23
6	20	6	168	24

Table 30. GIAI-202 Partition Table

Scheme	GIAI-202				
URI Template	`urn:epc:tag:giai-202:`*`F.C.A`*				
Total Bits	202				
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Individual Asset Reference
Logical Segment Bit Count	8	3	3	20-40	168–148
Coding Segment	EPC Header	Filter	GIAI		
URI portion		*`F`*	*`C.A`*		
Coding Segment Bit Count	8	3	191		
Bit Position	$b_{201}b_{200}\ldots b_{194}$	$b_{193}b_{192}b_{191}$	$b_{190}b_{189}\ldots b_{0}$		
Coding Method	00111000	Integer	String Partition Table 30		

2657 Table 31. GIAI-202 Coding Table

2658 14.5.6 Global Service Relation Number (GSRN)

2659 One coding scheme for the GSRN is specified: the 96-bit encoding GSRN-96. The
2660 GSRN-96 encoding allows for the full range of GSRN codes as specified in
2661 [GS1GS10.0].

2662 The GSRN-96 coding scheme makes reference to the following partition table.

Partition Value (*P*)	**Company Prefix**		**Service Reference**	
	Bits (*M*)	**Digits (*L*)**	**Bits (*N*)**	**Digits**
0	40	12	18	5
1	37	11	21	6
2	34	10	24	7
3	30	9	28	8

Partition Value (*P*)	Company Prefix		Service Reference	
	Bits (*M*)	**Digits (*L*)**	**Bits (*N*)**	**Digits**
4	27	8	31	9
5	24	7	34	10
6	20	6	38	11

Table 32. GSRN Partition Table

14.5.6.1 GSRN-96 Coding Table

Scheme	GSRN-96					
URI Template	`urn:epc:tag:gsrn-96:`*`F.C.S`*					
Total Bits	96					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Extension / Serial Reference	(Reserved)
Logical Segment Bit Count	8	3	3	20-40	38-18	24
Coding Segment	EPC Header	Filter	GSRN			(Reserved)
URI portion		*`F`*	*`C.S`*			
Coding Segment Bit Count	8	3	61			24
Bit Position	$b_{95}b_{94}\ldots b_{88}$	$b_{87}b_{86}b_{85}$	$b_{84}b_{83}\ldots b_{24}$			$b_{23}b_{22}\ldots b_0$
Coding Method	00101101	Integer	Partition Table 32			00…0 (24 zero bits)

Table 33. GSRN-96 Coding Table

14.5.7 Global Document Type Identifier (GDTI)

Two coding schemes for the GDTI specified, a 96-bit encoding (GDTI-96) and a 195-bit encoding (GDTI-113). The GDTI-113 encoding allows for the full range of document serial numbers up to 17 numeric characters (including leading zeros) as specified in [GS1GS10.0]. The GDTI-96 encoding allows for document serial numbers without leading zeros whose value is less than 2^{41} (that is, from 0 through 2,199,023,255,551, inclusive).

Only GDTIs that include the optional serial number may be represented as EPCs. A GDTI without a serial number represents a document class, rather than a specific document, and therefore may not be used as an EPC (just as a non-serialized GTIN may not be used as an EPC).

Both GDTI coding schemes make reference to the following partition table.

Partition Value (*P*)	**Company Prefix**		**Document Type**	
	Bits (*M*)	**Digits (*L*)**	**Bits (*N*)**	**Digits**
0	40	12	1	0
1	37	11	4	1
2	34	10	7	2
3	30	9	11	3
4	27	8	14	4
5	24	7	17	5
6	20	6	21	6

Table 34. GDTI Partition Table

2679 #### 14.5.7.1 GDTI-96 Coding Table

Scheme	GDTI-96					
URI Template	`urn:epc:tag:gdti-96:`*`F.C.D.S`*					
Total Bits	96					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Document Type	Serial
Logical Segment Bit Count	8	3	3	20-40	21-1	41
Coding Segment	EPC Header	Filter	Partition + Company Prefix + Document Type			Serial
URI portion		*F*	*C.D*			*S*
Coding Segment Bit Count	8	3	44			41
Bit Position	$b_{95}b_{94}\ldots b_{88}$	$b_{87}b_{86}b_{85}$	$b_{84}b_{83}\ldots b_{41}$			$b_{40}b_{39}\ldots b_{0}$
Coding Method	00101100	Integer	Partition Table 34			Integer

2680 Table 35. GDTI-96 Coding Table

14.5.7.2 GDTI-113 Coding Table

Scheme	GDTI-113					
URI Template	`urn:epc:tag:gdti-113:`*`F.C.D.S`*					
Total Bits	113					
Logical Segment	EPC Header	Filter	Partition	GS1 Company Prefix	Document Type	Serial
Logical Segment Bit Count	8	3	3	20-40	21-1	58
Coding Segment	EPC Header	Filter	Partition + Company Prefix + Document Type			Serial
URI portion		*`F`*	*`C.D`*			*`S`*
Coding Segment Bit Count	8	3	44			58
Bit Position	$b_{112}b_{111}\ldots b_{105}$	$b_{104}b_{103}b_{102}$	$b_{101}b_{100}\ldots b_{58}$			$b_{57}b_{56}\ldots b_0$
Coding Method	00111010	Integer	Partition Table 34			Numeric String

Table 36. GDTI-113 Coding Table

14.5.8 General Identifier (GID)

One coding scheme for the GID is specified: the 96-bit encoding GID-96. No partition table is required.

14.5.8.1 GID-96 Coding Table

Scheme	GID-96			
URI Template	`urn:epc:tag:gid-96:`*`M.C.S`*			
Total Bits	96			
Logical Segment	EPC Header	General Manager Number	Object Class	Serial Number
Logical Segment Bit Count	8	28	24	36
Coding Segment	EPC Header	General Manager Number	Object Class	Serial Number
URI portion		*`M`*	*`C`*	*`S`*
Coding Segment Bit Count	8	28	24	36
Bit Position	$b_{95}b_{94}\ldots b_{88}$	$b_{87}b_{86}\ldots b_{60}$	$b_{59}b_{58}\ldots b_{36}$	$b_{35}b_{34}\ldots b_{0}$
Coding Method	00110101	Integer	Integer	Integer

Table 37. GID-96 Coding Table

14.5.9 DoD Identifier

At the time of this writing, the details of the DoD encoding is explained in a document titled "United States Department of Defense Supplier's Passive RFID Information Guide" that can be obtained at the United States Department of Defense's web site (http://www.dodrfid.org/supplierguide.htm).

15 EPC Memory Bank Contents

This section specifies how to translate the EPC Tag URI and EPC Raw URI into the binary contents of the EPC memory bank of a Gen 2 Tag, and vice versa.

15.1 Encoding Procedures

This section specifies how to translate the EPC Tag URI and EPC Raw URI into the binary contents of the EPC memory bank of a Gen 2 Tag.

15.1.1 EPC Tag URI into Gen 2 EPC Memory Bank

Given:

- An EPC Tag URI beginning with `urn:epc:tag:`

Encoding procedure:

1. If the URI is not syntactically valid according to Section 12.4, stop: this URI cannot be encoded.
2. Apply the encoding procedure of Section 14.3 to the URI. The result is a binary string of *N* bits. If the encoding procedure fails, stop: this URI cannot be encoded.
3. Fill in the Gen 2 EPC Memory Bank according to the following table:

Bits	Field	Contents
00_h – $0F_h$	CRC	CRC code calculated from the remainder of the memory bank. (Normally, this is calculated automatically by the reader, and so software that implements this procedure need not be concerned with it.)
10_h – 14_h	Length	The number of bits, *N*, in the EPC binary encoding determined in Step 2 above, divided by 16, and rounded up to the next higher integer if *N* was not a multiple of 16.
15_h	User Memory Indicator	If the EPC Tag URI includes a control field `[umi=1]`, a one bit. If the EPC Tag URI includes a control field `[umi=0]` or does not contain a `umi` control field, a zero bit. Note that certain Gen 2 Tags may ignore the value written to this bit, and instead calculate the value of the bit from the contents of user memory. See [UHFC1G2].
16_h	XPC Indicator	This bit is calculated by the tag and ignored by the tag when the tag is written, and so is disregarded by this encoding procedure.
17_h	Toggle	0, indicating that the EPC bank contains an EPC
18_h – $1F_h$	Attribute Bits	If the EPC Tag URI includes a control field `[att=xNN]`, the value NN considered as an 8-bit hexadecimal number. If the EPC Tag URI does not contain such a control field, zero.
20_h – ?	EPC / UII	The *N* bits obtained from the EPCbinary encoding procedure in Step 2 above, followed by enough zero bits to bring the total number of bits to a multiple of 16 (0 – 15 extra zero bits)

Table 38. Recipe to Fill In Gen 2 EPC Memory Bank from EPC Tag URI

Explanation (non-normative): The XPC bits (bits 210_h – $21F_h$) are not included in this procedure, because the only XPC bits defined in [UHFC1G2] are bits which are written indirectly via recommissioning. Those bits are not intended to be written explicitly by an application.

15.1.2 EPC Raw URI into Gen 2 EPC Memory Bank

Given:

- An EPC Raw URI beginning with `urn:epc:raw:`. Such a URI has one of the following three forms:

  `urn:epc:raw:`*`OptionalControlFields`*`:`*`Length`*`.x`*`HexPayload`*

  `urn:epc:raw:`*`OptionalControlFields`*`:`*`Length`*`.x`*`AFI`*`.x`*`HexPayload`*

  `urn:epc:raw:`*`OptionalControlFields`*`:`*`Length`*`.`*`DecimalPayload`*

Encoding procedure:

1. If the URI is not syntactically valid according to the grammar in Section 12.4, stop: this URI cannot be encoded.
2. Extract the leftmost `NonZeroComponent` according to the grammar (the *`Length`* field in the templates above). This component immediately follows the rightmost colon (`:`) character. Consider this as a decimal integer, *N*. This is the number of bits in the raw payload.
3. Determine the toggle bit and AFI (if any):
   - 3.1. If the body of the URI matches the `DecimalRawURIBody` or `HexRawURIBody` production of the grammar (the first and third templates above), the toggle bit is zero.
   - 3.2. If the body of the URI matches the `AFIRawURIBody` production of the grammar (the second template above), the toggle bit is one. The AFI is the value of the leftmost `HexComponent` within the `AFIRawURIBody` (the *`AFI`* field in the template above), considered as an 8-bit unsigned hexadecimal integer. If the value of the `HexComponent` is greater than or equal to 256, stop: this URI cannot be encoded.
4. Determine the EPC/UII payload:
   - 4.1. If the body of the URI matches the `HexRawURIBody` production of the grammar (first template above) or `AFIRawURIBody` production of the grammar (second template above), the payload is the rightmost `HexComponent` within the body (the *`HexPayload`* field in the templates above), considered as an *N*-bit unsigned hexadecimal integer, where *N* is as determined in Step 2 above. If the value of this `HexComponent` greater than or equal to 2^N, stop: this URI cannot be encoded.
   - 4.2. If the body of the URI matches the `DecimalRawURIBody` production of the grammar (third template above), the payload is the rightmost `NumericComponent` within the body (the *`DecimalPayload`* field in the template above), considered as an *N*-bit unsigned decimal integer, where *N* is as determined in Step 2 above. If the value of this `NumericComponent` greater than or equal to 2^N, stop: this URI cannot be encoded.
5. Fill in the Gen 2 EPC Memory Bank according to the following table:

Bits	Field	Contents
00_h – $0F_h$	CRC	CRC code calculated from the remainder of the memory bank. (Normally, this is calculated automatically by the reader, and so software that implements this procedure need not be concerned with it.)
10_h – 14_h	Length	The number of bits, *N*, in the EPC binary encoding determined in Step 2 above, divided by 16, and rounded up to the next higher integer if *N* was not a multiple of 16.
15_h	User Memory Indicator	This bit is calculated by the tag and ignored by the tag when the tag is written, and so is disregarded by this encoding procedure.
16_h	XPC Indicator	This bit is calculated by the tag and ignored by the tag when the tag is written, and so is disregarded by this encoding procedure.
17_h	Toggle	The value determined in Step 3, above.
18_h – $1F_h$	AFI / Attribute Bits	If the toggle determined in Step 3 is one, the value of the AFI determined in Step 3.2. Otherwise, If the URI includes a control field `[att=xNN]`, the value NN considered as an 8-bit hexadecimal number. If the URI does not contain such a control field, zero.
20_h – ?	EPC / UII	The *N* bits determined in Step 4 above, followed by enough zero bits to bring the total number of bits to a multiple of 16 (0 – 15 extra zero bits)

Table 39. Recipe to Fill In Gen 2 EPC Memory Bank from EPC Raw URI

15.2 Decoding Procedures

This section specifies how to translate the binary contents of the EPC memory bank of a Gen 2 Tag into the EPC Tag URI and EPC Raw URI.

15.2.1 Gen 2 EPC Memory Bank into EPC Raw URI

Given:

- The contents of the EPC Memory Bank of a Gen 2 tag

Procedure:

1. Extract the length bits, bits 10_h – 14_h. Consider these bits to be an unsigned integer *L*.
2. Calculate *N* = 16*L*.
3. If bit 17_h is set to one, extract bits 18_h – $1F_h$ and consider them to be an unsigned integer *A*. Construct a string consisting of the letter "`x`", followed by *A* as a 2-digit

hexadecimal numeral (using digits and uppercase letters only), followed by a period ("`.`").

4. Apply the decoding procedure of Section 0 to decode control fields.
5. Extract *N* bits beginning at bit 20_h and consider them to be an unsigned integer *V*. Construct a string consisting of the letter "`x`" followed by *V* as a (*N*/4)-digit hexadecimal numeral (using digits and uppercase letters only).
6. Construct a string consisting of "`urn:epc:raw:`", followed by the result from Step 4 (if not empty), followed by *N* as a decimal numeral without leading zeros, followed by a period ("`.`"), followed by the result from Step 3 (if not empty), followed by the result from Step 5. This is the final EPC Raw URI.

➢ Grammar issue for zero length bits

15.2.2 Gen 2 EPC Memory Bank into EPC Tag URI

This procedure decodes the contents of a Gen 2 EPC Memory bank into an EPC Tag URI beginning with `urn:epc:tag:` if the memory contains a valid EPC, or into an EPC Raw URI beginning `urn:epc:raw:` otherwise.

Given:

- The contents of the EPC Memory Bank of a Gen 2 tag

Procedure:

1. Extract the length bits, bits 10_h – 14_h. Consider these bits to be an unsigned integer *L*.
2. Calculate $N = 16L$.
3. Extract *N* bits beginning at bit 20_h. Apply the decoding procedure of Section 14.4, passing the *N* bits as the input to that procedure.
4. If the decoding procedure of Section 14.4 fails, continue with the decoding procedure of Section 15.2.1 to compute an EPC Raw URI. Otherwise, the decoding procedure of of Section 14.4 yielded an EPC Tag URI beginning `urn:epc:tag:`. Continue to the next step.
5. Apply the decoding procedure of Section 0 to decode control fields.
6. Insert the result from Section 0 (including any trailing colon) into the EPC Tag URI obtained in Step 4, immediately following the `urn:epc:tag:` prefix. (If Section 0 yielded an empty string, this result is identical to what was obtained in Step 4.) The result is the final EPC Tag URI.

➢ What about partial tag write – see existing tds

15.2.3 Gen 2 EPC Memory Bank into Pure Identity EPC URI

This procedure decodes the contents of a Gen 2 EPC Memory bank into a Pure Identity EPC URI beginning with `urn:epc:id:` if the memory contains a valid EPC, or into an EPC Raw URI beginning `urn:epc:raw:` otherwise.

Given:

- The contents of the EPC Memory Bank of a Gen 2 tag

Procedure:

1. Apply the decoding procedure of Section 15.2.2 to obtain either an EPC Tag URI or an EPC Raw URI. If an EPC Raw URI is obtained, this is the final result.
2. Otherwise, apply the procedure of Section 12.3.3 to the EPC Tag URI from Step 1 to obtain a Pure Identity EPC URI. This is the final result.

15.2.4 Decoding of Control Information

This procedure is used as a subroutine by the decoding procedures in Sections 15.2.1 and 15.2.2. It calculates a string that is inserted immediately following the `urn:epc:tag:` or `urn:epc:raw:` prefix, containing the values of all non-zero control information fields (apart from the filter value). If all such fields are zero, this procedure returns an empty string, in which case nothing additional is inserted after the `urn:epc:tag:` or `urn:epc:raw:` prefix.

Given:

- The contents of the EPC Memory Bank of a Gen 2 tag

Procedure:

1. If bit 17_h is zero, extract bits 18_h – $1F_h$ and consider them to be an unsigned integer *A*. If *A* is non-zero, append the string `[att=x`*`AA`*`]` (square brackets included) to *CF*, where *AA* is the value of *A* as a two-digit hexadecimal numeral.
2. If bit 15_h is non-zero, append the string `[umi=1]` (square brackets included) to *CF*.
3. If bit 16_h is non-zero, extract bits 210_h – $21F_h$ and consider them to be an unsigned integer *X*. If *X* is non-zero, append the string `[xpc=x`*`XXXX`*`]` (square brackets included) to *CF*, where *XXXX* is the value of *X* as a four-digit hexadecimal numeral. Note that in the Gen 2 air interface, bits 210_h – $21F_h$ are inserted into the backscattered inventory data immediately following bit $1F_h$, when bit 16_h is non-zero. See [UHFC1G2].
4. Return the resulting string (which may be empty).

16 Tag Identification (TID) Memory Bank Contents

To conform to this specification, the Tag Identification memory bank (bank 10) SHALL contain an 8 bit ISO/IEC 15963 allocation class identifier of $E2_h$ at memory locations 00_h to 07_h. TID memory locations 08_h to 13_h SHALL contain a 12 bit Tag mask designer identifier (MDID) obtainable from EPCglobal. TID memory locations 14_h to $1F_h$ SHALL contain a 12-bit vendor-defined Tag model number (TMN) as described below.

EPCglobal will assign two MDIDs to each mask designer, one with bit 08_h equal to one and one with bit 08_h equal to zero. Readers and applications that are not configured to handle the extended TID will treat both of these numbers as a 12 bit MDID. Readers and applications that are configured to handle the extended TID will recognize the TID memory location 08_h as the Extended Tag Identification bit. The value of this bit indicates the format of the rest of the TID. A value of zero indicates a short TID in which the values beyond address $1F_h$ are not defined. A value of one indicates an Extended Tag Identification (XTID) in which the memory locations beyond $1F_h$ contain additional data as specified in Section 16.2.

The Tag model number (TMN) may be assigned any value by the holder of a given MDID. However, [UHFC1G2] states "TID memory locations above 07_h shall be defined according to the registration authority defined by this class identifier value and shall contain, at a minimum, sufficient identifying information for an Interrogator to uniquely identify the custom commands and/or optional features that a Tag supports." For the allocation class identifier of $E2_h$ this information is the MDID and TMN, regardless of whether the extended TID is present or not. If two tags differ in custom commands and/or optional features, they must be assigned different MDID/TMN combinations. In particular, if two tags contain an extended TID and the values in their respective extended TIDs differ in any value other than the value of the serial number, they must be assigned a different MDID/TMN combination. (The serial number by definition must be different for any two tags having the same MDID and TMN, so that the Serialized Tag Identification specified in Section 16.3 is globally unique.) For tags that do not contain an extended TID, it should be possible in principle to use the MDID and TMN to look up the same information that would be encoded in the extended TID were it actually present on the tag, and so again a different MDID/TMN combination must be used if two tags differ in the capabilities as they would be described by the extended TID, were it actually present.

16.1 Short Tag Identification

If the XTID bit (bit 08_h of the TID bank) is set to zero, the TID bank only contains the allocation class identifier, mask designer identifier (MDID), and Tag model number (TMN) as specified above. Readers and applications that are not configured to handle the extended TID will treat all TIDs as short tag identification, regardless of whether the XTID bit is zero or one.

Note: The memory maps depicted in this document are identical to how they are depicted in [UHFC1G2]. The lowest word address starts at the bottom of the map and increases as you go up the map. The bit address reads from left to right starting with bit zero and ending with bit fifteen. The fields (MDID, TMN, etc) described in the document put their most significant bit (highest bit number) into the lowest bit address in memory and the least significant bit (bit zero) into the highest bit address in memory. Take the ISO/IEC 15963 allocation class identifier of $E2_h = 11100010_2$ as an example. The most significant bit of this field is a one and it resides at address 00_h of the TID memory bank. The least significant bit value is a zero and it resides at address 07_h of the TID memory bank. When tags backscatter data in response to a read command they transmit each word starting from bit address zero and ending with bit address fifteen.

TID MEM BANK BIT ADDRESS	BIT ADDRESS WITHIN WORD (In Hexadecimal)															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10_h-$1F_h$	TAG MDID[3:0]				TAG MODEL NUMBER[11:0]											
00_h-$0F_h$	$E2_h$								TAG MDID[11:4]							

Table 40. Short TID format

16.2 Extended Tag Identification (XTID)

The XTID is intended to provide more information to end users about the capabilities of tags that are observed in their RFID applications. The XTID extends the format by adding support for serialization and information about key features implemented by the tag.

If the XTID bit (bit 08_h of the TID bank) is set to one, the TID bank SHALL contain the allocation class identifier, mask designer identifier (MDID), and Tag model number (TMN) as specified above, and SHALL also contain additional information as specified in this section.

TID memory locations 20_h to $2F_h$ SHALLcontain a 16-bit XTID header as specified in Section 16.2.1. The values in the XTID header specify what additional information is present in memory locations 30_h and above. TID memory locations 00_h through $2F_h$ are the only fixed location fields in the extended TID; all fields following the XTID header can vary in their location in memory depending on the values in the XTID header.

The information in the XTID following the XTID header SHALL consist of zero or more multi-word "segments," each segment being divided into one or more "fields," each field providing certain information about the tag as specified below. The XTID header indicates which of the XTID segments the tag mask-designer has chosen to include. The order of the XTID segments in the TID bank shall follow the order that they are listed in the XTID header from most significant bit to least significant bit. If an XTID segment is not present then segments at less significant bits in the XTID header shall move to lower TID memory addresses to keep the XTID memory structure contiguous. In this way a minimum amount of memory is used to provide a serial number and/or describe the features of the tag. A fully populated XTID is shown in the table below.

Informative: The XTID header corresponding to this memory map would be 0011110000000000_2. If the tag only contained a 48 bit serial number the XTID header would be 0010000000000000_2. The serial number would start at bit address 30_h and end at bit address $5F_h$. If the tag contained just the BlockWrite and BlockErase segment and the User Memory and BlockPermaLock segment the XTID header would be 0000110000000000_2. The BlockWrite and BlockErase segment would start at bit address 30_h and end at bit address $6F_h$. The User Memory and BlockPermaLock segment would start at bit address 70_h and end at bit address $8F_h$.

TDS	TID MEM	BIT ADDRESS WITHIN WORD (In Hexadecimal)

Reference Section	BANK BIT ADDRESS	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
16.2.5	C0$_h$-CF$_h$	**User Memory and BlockPermaLock Segment [15:0]**															
	B0$_h$-BF$_h$	**User Memory and BlockPermaLock Segment [31:16]**															
16.2.4	A0$_h$-AF$_h$	**BlockWrite and BlockErase Segment [15:0]**															
	90$_h$-9F$_h$	**BlockWrite and BlockErase Segment [31:16]**															
	80$_h$-8F$_h$	**BlockWrite and BlockErase Segment [47:32]**															
	70$_h$-7F$_h$	**BlockWrite and BlockErase Segment [63:48]**															
16.2.3	60$_h$-6F$_h$	**Optional Command Support Segment [15:0]**															
16.2.2	50$_h$-5F$_h$	**Serial Number Segment [15:0]**															
	40$_h$-4F$_h$	**Serial Number Segment [31:16]**															
	30$_h$-3F$_h$	**Serial Number Segment [47:32]**															
16.2.1	20$_h$-2F$_h$	**XTID Header Segment [15:0]**															
16.1 and 16.2	10$_h$-1F$_h$	**TAG MDID[3:0]**				**TAG MODEL NUMBER[11:0]**											
	00$_h$-0F$_h$	**E2$_h$**								**TAG MDID[11:4]**							

Table 41. The Extended Tag Identification (XTID) format for the TID memory bank. Note that the table above is fully filled in and that the actual amount of memory used, presence of a segment, and address location of a segment depends on the XTID Header.

16.2.1 XTID Header

The XTID header is shown in Table 42. It contains defined and reserved for future use (RFU) bits. The extended header bit and RFU bits (bits 9 through 0) shall be set to zero to comply with this version of the specification. Bits 15 through 13 of the XTID header word indicate the presence and size of serialization on the tag. If they are set to zero then there is no serialization in the XTID. If they are not zero then there is a tag serial number immediately following the header. The optional features currently in bits 12 through 10 are handled differently. A zero indicates the reader needs to perform a database look up or that the tag does not support the optional feature. A one indicates that the tag supports the optional feature and that the XTID contains the segment describing this feature

Bit Position in Word	Field	Description
0	Extended Header Present	If non-zero, specifies that additional XTID header bits are present beyond the 16 XTID header bits specified herein. This provides a mechanism to extend the XTID in future versions of the EPC Tag Data Standard. This bit SHALL be set to zero to comply with this version of the EPC Tag Data Standard. If zero, specifies that the XTID header only contains the 16 bits defined herein.
9 – 1	RFU	Reserved for future use. These bits SHALL be zero to comply with this version of the EPC Tag Data Standard
10	User Memory and Block Perma Lock Segment Present	If non-zero, specifies that the XTID includes the User Memory and Block PermaLock segment specified in Section 16.2.5. If zero, specifies that the XTID does not include the User Memory and Block PermaLock words.
11	BlockWrite and BlockErase Segment Present	If non-zero, specifies that the XTID includes the BlockWrite and BlockErase segment specified in Section 16.2.4. If zero, specifies that the XTID does not include the BlockWrite and BlockErase words.
12	Optional Command Support Segment Present	If non-zero, specifies that the XTID includes the Optional Command Support segment specified in Section 16.2.3. If zero, specifies that the XTID does not include the Optional Command Support word.
13 – 15	Serialization	If non-zero, specifies that the XTID includes a unique serial number, whose length in bits is $48 + 16(N-1)$, where N is the value of this field. If zero, specifies that the XTID does not include a unique serial number.

Table 42. The XTID header

16.2.2 XTID Serialization

The length of the XTID serialization is specified in the XTID header. The managing entity specified by the tag mask designer ID is responsible for assigning unique serial numbers for each tag model number. The length of the serial number uses the following algorithm:

0: Indicates no serialization

1-7: Length in bits = 48 + ((Value-1) * 16)

16.2.3 Optional Command Support Segment

If bit twelve is set in the XTID header then the following word is added to the XTID. Bit fields that are left as zero indicate that the tag does not support that feature. The description of the features is as follows.

Bit Position in Segment	Field	Description
4 – 0	Max EPC Size	This five bit field shall indicate the maximum size that can be programmed into the first five bits of the PC.
5	Recom Support	If this bit is set the tag supports recommissioning as specified in [UHFC1G2].
6	Access	If this bit is set the it indicates that the tag supports the access command.
7	Separate Lockbits	If this bit is set it means that the tag supports lock bits for each memory bank rather than the simplest implementation of a single lock bit for the entire tag.
8	Auto UMI Support	If this bit is set it means that the tag automatically sets its user memory indicator bit in the PC word.
9	PJM Support	If this bit is set it indicates that the tag supports phase jitter modulation. This is an optional modulation mode supported only in Gen 2 HF tags.
10	BlockErase Supported	If set this indicates that the tag supports the BlockErase command. How the tag supports the BlockErase command is described in Section 16.2.4. A manufacture may choose to set this bit, but not include the BlockWrite and BlockErase field if how to use the command needs further explanation through a database lookup.
11	BlockWrite Supported	If set this indicates that the tag supports the BlockWrite command. How the tag supports the BlockErase command is described in Section 16.2.4. A manufacture may choose to set this bit, but not include the BlockWrite and BlockErase field if how to use the command needs further explanation through a database lookup.

Bit Position in Segment	Field	Description
12	BlockPermaLock Supported	If set this indicates that the tag supports the BlockPermaLock command. How the tag supports the BlockPermaLock command is described in Section 16.2.5. A manufacture may choose to set this bit, but not include the BlockPermaLock and User Memory field if how to use the command needs further explanation through a database lookup.
15 – 13	[RFU]	These bits are RFU and should be set to zero.

Table 43. Optional Command Support XTID Word

16.2.4 BlockWrite and BlockErase Segment

If bit eleven of the XTID header is set then the XTID shall include the four-word BlockWrite and BlockErase segment. To indicate that a command is not supported, the tag shall have all fields related to that command set to zero. The descriptions of the fields are as follows.

Bit Position in Segment	Field	Description
7 – 0	Block Write Size	Max block size that the tag supports for the BlockWrite command. This value should be between 1-255 if the BlockWrite command is described in this field.
8	Variable Size Block Write	This bit is used to indicate if the tag supports BlockWrite commands with variable sized blocks. • If the value is zero the tag only supports writing blocks exactly the maximum block size indicated in bits [7-0]. • If the value is one the tag supports writing blocks less than the maximum block size indicated in bits [7-0].
16 – 9	Block Write EPC Address Offset	This indicates the starting word address of the first full block that may be written to using BlockWrite in the EPC memory bank.

Bit Position in Segment	Field	Description
17	No Block Write EPC address alignment	This bit is used to indicate if the tag memory architecture has hard block boundaries in the EPC memory bank. • If the value is zero the tag has hard block boundaries in the EPC memory bank. The tag will not accept BlockWrite commands that start in one block and end in another block. These block boundaries are determined by the max block size and the starting address of the first full block. All blocks have the same maximum size. • If the value is one the tag has no block boundaries in the EPC memory bank. It will accept all BlockWrite commands that are within the memory bank.
25 – 18	Block Write User Address Offset	This indicates the starting word address of the first full block that may be written to using BlockWrite in the User memory.
26	No Block Write User Address Alignment	This bit is used to indicate if the tag memory architecture has hard block boundaries in the USER memory bank. • If the value is zero the tag has hard block boundaries in the USER memory bank. The tag will not accept BlockWrite commands that start in one block and end in another block. These block boundaries are determined by the max block size and the starting address of the first full block. All blocks have the same maximum size. • If the value is one the tag has no block boundaries in the USER memory bank. It will accept all BlockWrite commands that are within the memory bank.
31 – 27	[RFU]	These bits are RFU and should be set to zero.
39 –32	Size of Block Erase	Max block size that the tag supports for the BlockErase command. This value should be between 1-255 if the BlockErase command is described in this field.

<table>
<tr><th>Bit Position in Segment</th><th>Field</th><th>Description</th></tr>
<tr><td>40</td><td>Variable Size Block Erase</td><td>This bit is used to indicate if the tag supports BlockErase commands with variable sized blocks.
• If the value is zero the tag only supports erasing blocks exactly the maximum block size indicated in bits [39-32].
• If the value is one the tag supports erasing blocks less than the maximum block size indicated in bits [39-32].</td></tr>
<tr><td>48 – 41</td><td>Block Erase EPC Address Offset</td><td>This indicates the starting address of the first full block that may be erased in EPC memory bank.</td></tr>
<tr><td>49</td><td>No Block Erase EPC Address Alignment</td><td>This bit is used to indicate if the tag memory architecture has hard block boundaries in the EPC memory bank.
• If the value is zero the tag has hard block boundaries in the EPC memory bank. The tag will not accept BlockErase commands that start in one block and end in another block. These block boundaries are determined by the max block size and the starting address of the first full block. All blocks have the same maximum size.
• If the value is one the tag has no block boundaries in the EPC memory bank. It will accept all BlockErase commands that are within the memory bank.</td></tr>
<tr><td>57 – 50</td><td>Block Erase User Address Offset</td><td>This indicates the starting address of the first full block that may be erased in User memory bank.</td></tr>
</table>

Bit Position in Segment	Field	Description
58	No Block Erase User Address Alignment	Bit 58: This bit is used to indicate if the tag memory architecture has hard block boundaries in the USER memory bank. • If the value is zero the tag has hard block boundaries in the USER memory bank. The tag will not accept BlockErase commands that start in one block and end in another block. These block boundaries are determined by the max block size and the starting address of the first full block. All blocks have the same maximum size. • If the value is one the tag has no block boundaries in the USER memory bank. It will accept all BlockErase commands that are within the memory bank.
63 – 59	[RFU]	These bits are reserved for future use and should be set to zero.

Table 44. XTID Block Write and Block Erase Information

16.2.5 User Memory and BlockPermaLock Segment

This two-word segment is present in the XTID if bit 10 of the XTID header is set. Bits 15-0 shall indicate the size of user memory in words. Bits 31-16 shall indicate the size of the blocks in the USER memory bank in words for the BlockPermaLock command. Note: These block sizes only apply to the BlockPermaLock command and are independent of the BlockWrite and BlockErase commands.

Bit Position in Segment	Field	Description
15 – 0	User Memory Size	Number of 16-bit words in user memory.
31 –16	BlockPermaLock Block Size	If non-zero, the size in words of each block that may be block permalocked. That is, the block permalock feature allows blocks of *N**16 bits to be locked, where *N* is the value of this field. If zero, then the XTID does not describe the block size for the BlockPermaLock feature. The tag may or may not support block permalocking.

Table 45. XTID Block PermaLock and User Memory Information

16.3 Serialized Tag Identification (STID)

This section specifies a URI form for the serialization encoded within an XTID, called the Serialized Tag Identifier (STID). The STID URI form may be used by business applications that use the serialized TID to uniquely identify the tag onto which an EPC has been programmed. The STID URI is intended to supplement, not replace, the EPC for those applications that make use of RFID tag serialization in addition to the EPC that uniquely identifies the physical object to which the tag is affixed; e.g., in an application that uses the STID to help ensure a tag has not been counterfeited.

16.3.1 STID URI Grammar

The syntax of the STID URI is specified by the following grammar:

```
STID-URI ::= "urn:epc:stid:" 2*( "x" HexComponent "." ) "x"
HexComponent
```

where the first and second `HexComponents` SHALL consist of exactly three `UpperHexChar`s and the third `HexComponent` SHALL consist of 12, 16, 20, 24, 28, 32, or 36 `UpperHexChar`s.

The first `HexComponent` is the value of the Tag Mask Designer ID (MDID) as specified in Sections 16.1 and 16.2. The second `HexComponent` is the value of the Tag Model Number as specified in Sections 16.1 and 16.2. The third `HexComponent` is the value of the XTID serial number as specified in Sections 16.2 and 16.2.2. The number of `UpperHexChar`s in the third `HexComponent` is equal to the number of bits in the XTID serial number divided by four.

16.3.2 Decoding Procedure: TID Bank Contents to STID URI

The following procedure specifies how to construct an STID URI given the contents of the TID bank of a Gen 2 Tag.

Given:

- The contents of the TID memory bank of a Gen 2 Tag, as a bit string $b_0b_1\ldots b_{N-1}$, where the number of bits N is at least 48.

Yields:

- An STID-URI

Procedure:

1. Bits $b_0\ldots b_7$ should match the value 11100010. If not, stop: this TID bank contents does not contain an XTID as specified herein.
2. Bit b_8 should be set to one. If not, stop: this TID bank contents does not contain an XTID as specified herein.
3. Consider bits $b_8\ldots b_{19}$ as a 12 bit unsigned integer. This is the Tag Mask Designer ID (MDID).
4. Consider bits $b_{20}\ldots b_{31}$ as a 12 bit unsigned integer. This is the Tag Model Number.

5. Consider bits $b_{32}…b_{34}$ as a 3-bit unsigned integer V. If V equals zero, stop: this TID bank contents does not contain a serial number. Otherwise, calculate the length of the serial number L = 48 + 16(V − 1). Consider bits $b_{48}b_{49}…b_{48+L-1}$ as an L-bit unsigned integer. This is the serial number.

6. Construct the STID-URI by concatenating the following strings: the prefix `urn:epc:stid:`, the lowercase letter `x`, the value of the MDID from Step 3 as a 3-character hexadecimal numeral, a dot (`.`) character, the lowercase letter `x`, the value of the Tag Model Number from Step 4 as a 3-character hexadecimal numeral, a dot (`.`) character, the lowercase letter `x`, and the value of the serial number from Step 5 as a (L/4)-character hexadecimal numeral. Only uppercase letters `A` through `F` shall be used in constructing the hexadecimal numerals.

17 User Memory Bank Contents

The EPCglobal User Memory Bank provides a variable size memory to store additional data attributes related to the object identified in the EPC Memory Bank of the tag.

User memory may or may not be present on a given tag. When user memory is not present, bit 15_h of the EPC memory bank SHALL be set to zero. When user memory is present and uninitialized, bit 15_h of the EPC memory bank SHALL be set to zero and bits 03_h through 07_h of the User Memory bank SHALL be set to zero. When user memory is present and initialized, bit 15_h of the Protocol Control Word in EPC memory SHALL be set to one to indicate the presence of encoded data in User Memory, and the user memory bank SHALL be programmed as specified herein.

To conform with this specification, the first eight bits of the User Memory Bank SHALL contain a Data Storage Format Identifier (DSFID) as specified in [ISO15962]. This maintains compatibility with other standards. The DSFID consists of three logical fields: Access Method, Extended Syntax Indicator, and Data Format. The Access Method is specified in the two most significant bits of the DSFID, and is encoded with the value "10" to designate the "Packed Objects" Access Method as specified in Appendix I herein if the "Packed Objects" Access Method is employed, and is encoded with the value "00" to designate the "No-Directory" Access Method as specified in [ISO15962] if the "No-Directory" Access Method is employed. The next bit is set to one if there is a second DSFID byte present. The five least significant bits specify the Data Format, which indicates what data system predominates in the memory contents. If GS1 Application Identifiers (AIs) predominate, the value of "01001" specifies the GS1 Data Format 09 as registered with ISO, which provides most efficient support for the use of AI data elements. Appendix I through Appendix M of this specification contain the complete specification of the "Packed Objects" Access Method; it is expected that this content will appear as Annex I through Annex M, respectively, of ISO/IEC 15962, 2nd Edition [ISO15962], when the latter becomes available A complete definition of the DSFID is specified in ISO/IEC 15962 [ISO15962]. A complete definition of the table that governs the Packed Objects encoding of Application Identifiers (AIs) is specified by GS1 and registered with ISO under the procedures of ISO/IEC 15961, and is reproduced in Appendix F. This table is similar in format to the hypothetical example shown as Table

L-1 in Appendix L, but with entries to accommodate encoding of all valid Application Identifiers.

A tag whose User Memory Bank programming conforms to this specification SHALL be encoded using either the Packed Objects Access Method or the No-Directory Access Method, provided that if the No-Directory Access Method is used that the “application-defined” compaction mode as specified in [ISO15962] SHALL NOT be used. A tag whose User Memory Bank programming conforms to this specification MAY use any registered Data Format including Data Format 09.

Where the Packged Objects specification in Appendix I makes reference to Extensible Bit Vectors (EBVs), the format specified in Appendix D SHALL be used.

A hardware or software component that conforms to this specification for User Memory Bank reading and writing SHALL fully implement the Packed Objects Access Method as specified in Appendix I through Appendix M of this specification (implying support for all registered Data Formats), SHALL implement the No-Directory Access Method as specified in [ISO15962], and MAY implement other Access Methods defined in [ISO15962] and subsequent versions of that standard. A hardware or software component NEED NOT, however, implement the“application-defined” compaction mode of the No-Directory Access Method as specified in [ISO15962]. A hardware or software component whose intended function is only to initialize tags (e.g., a printer) may conform to a subset of this specification by implementing either the Packed Objects or the No-Directory access method, but in this case NEED NOT implement both.

Explanation (non-normative): This specification allows two methods of encoding data in user memory. The ISO/IEC 15962 “No-Directory” Access Method has an installed base owing to its longer history and acceptance within certain end user communities. The Packed Objects Access Method was developed to provide for more efficient reading and writing of tags, and less tag memory consumption.

The “application-defined” compaction mode of the No-Directory Access Method is not allowed because it cannot be understood by a receiving system unless both sides have the same definition of how the compaction works.

Note that the Packed Objects Access Method supports the encoding of data either with or without a directory-like structure for random access. The fact that the other access method is named “No-Directory” in [ISO15962] should not be taken to imply that the Packed Objects Access Method always includes a directory.

Appendix A Character Set for Alphanumeric Serial Numbers

The following table specifies the characters that are permitted by the GS1 General Specifications [GS1GS10.0] for use in alphanumeric serial numbers. The columns are as follows:

- *Graphic Symbol* The printed representation of the character as used in human-readable forms.
- *Name* The common name for the character

- *Hex Value* A hexadecimal numeral that gives the 7-bit binary value for the character as used in EPC binary encodings. This hexadecimal value is always equal to the ISO 646 (ASCII) code for the character.
- *URI Form* The representation of the character within Pure Identity EPC URI and EPC Tag URI forms. This is either a single character whose ASCII code is equal to the value in the “hex value” column, or an escape triplet consisting of a percent character followed by two characters giving the hexadecimal value for the character.

Graphic Symbol	**Name**	**Hex Value**	**URI Form**	**Graphic Symbol**	**Name**	**Hex Value**	**URI Form**
`!`	Exclamation Mark	21	`!`	`M`	Capital Letter M	4D	`M`
`"`	Quotation Mark	22	`%22`	`N`	Capital Letter N	4E	`N`
`%`	Percent Sign	25	`%25`	`O`	Capital Letter O	4F	`O`
`&`	Ampersand	26	`%26`	`P`	Capital Letter P	50	`P`
`'`	Apostrophe	27	`'`	`Q`	Capital Letter Q	51	`Q`
`(`	Left Parenthesis	28	`(`	`R`	Capital Letter R	52	`R`
`)`	Right Parenthesis	29	`)`	`S`	Capital Letter S	53	`S`
`*`	Asterisk	2A	`*`	`T`	Capital Letter T	54	`T`
`+`	Plus sign	2B	`+`	`U`	Capital Letter U	55	`U`
`,`	Comma	2C	`,`	`V`	Capital Letter V	56	`V`
`-`	Hyphen/ Minus	2D	`-`	`W`	Capital Letter W	57	`W`
`.`	Full Stop	2E	`.`	`X`	Capital Letter X	58	`X`
`/`	Solidus	2F	`%2F`	`Y`	Capital Letter Y	59	`Y`
`0`	Digit Zero	30	`0`	`Z`	Capital Letter Z	5A	`Z`

Graphic Symbol	Name	Hex Value	URI Form	Graphic Symbol	Name	Hex Value	URI Form
1	Digit One	31	1	_	Low Line	5F	_
2	Digit Two	32	2	a	Small Letter a	61	a
3	Digit Three	33	3	b	Small Letter b	62	b
4	Digit Four	34	4	c	Small Letter c	63	c
5	Digit Five	35	5	d	Small Letter d	64	d
6	Digit Six	36	6	e	Small Letter e	65	e
7	Digit Seven	37	7	f	Small Letter f	66	f
8	Digit Eight	38	8	g	Small Letter g	67	g
9	Digit Nine	39	9	h	Small Letter h	68	h
:	Colon	3A	:	i	Small Letter i	69	i
;	Semicolon	3B	;	j	Small Letter j	6A	j
<	Less-than Sign	3C	%3C	k	Small Letter k	6B	k
=	Equals Sign	3D	=	l	Small Letter l	6C	l
>	Greater-than Sign	3E	%3E	m	Small Letter m	6D	m
?	Question Mark	3F	%3F	n	Small Letter n	6E	n
A	Capital Letter A	41	A	o	Small Letter o	6F	o
B	Capital Letter B	42	B	p	Small Letter p	70	p

Graphic Symbol	Name	Hex Value	URI Form	Graphic Symbol	Name	Hex Value	URI Form
C	Capital Letter C	43	C	q	Small Letter q	71	q
D	Capital Letter D	44	D	r	Small Letter r	72	r
E	Capital Letter E	45	E	s	Small Letter s	73	s
F	Capital Letter F	46	F	t	Small Letter t	74	t
G	Capital Letter G	47	G	u	Small Letter u	75	u
H	Capital Letter H	48	H	v	Small Letter v	76	v
I	Capital Letter I	49	I	w	Small Letter w	77	w
J	Capital Letter J	4A	J	x	Small Letter x	78	x
K	Capital Letter K	4B	K	y	Small Letter y	79	y
L	Capital Letter L	4C	L	z	Small Letter z	7A	z

Table 46. Characters Permitted in Alphanumeric Serial Numbers

Appendix B Glossary (non-normative)

Term	Defined Where	Meaning
Application Identifier (AI)	[GS1GS10.0]	A numeric code that identifies a data element within a GS1 Element String.
Attribute Bits	Section 11	An 8-bit field of control information that is stored in the EPC Memory Bank of a Gen 2 RFID Tag when the tag contains an EPC. The Attribute Bits includes data that guides the handling of the object to which the tag is affixed, for example a bit that indicates the presence of hazardous material.
Bar Code		A data carrier that holds text data in the form of light and dark markings which may be read by an optical reader device.

Term	Defined Where	Meaning
Control Information	Section 9.1	Information that is used by data capture applications to help control the process of interacting with RFID Tags. Control Information includes data that helps a capturing application filter out tags from large populations to increase read efficiency, special handling information that affects the behavior of capturing application, information that controls tag security features, and so on. Control Information is typically *not* passed directly to business applications, though Control Information may influence how a capturing application presents business data to the business application level. Unlike Business Data, Control Information has no equivalent in bar codes or other data carriers.
Data Carrier		Generic term for a marking or device that is used to physically attach data to a physical object. Examples of data carriers include Bar Codes and RFID Tags.
Electronic Product Code (EPC)	Section 4	A universal identifier for any physical object. The EPC is designed so that every physical object of interest to information systems may be given an EPC that is globally unique and persistent through time. The primary representation of an EPC is in the form of a Pure Identity EPC URI (*q.v.*), which is a unique string that may be used in information systems, electronic messages, databases, and other contexts. A secondary representation, the EPC Binary Encoding (*q.v.*) is available for use in RFID Tags and other settings where a compact binary representation is required.
EPC	Section 4	See Electronic Product Code
EPC Bank (of a Gen 2 RFID Tag)	[UHFC1G2]	Bank 01 of a Gen 2 RFID Tag as specified in [UHFC1G2]. The EPC Bank holds the EPC Binary Encoding of an EPC, together with additional control information as specified in Section 8.

Term	Defined Where	Meaning
EPC Binary Encoding	Section 13	A compact encoding of an Electronic Product Code, together with a filter value (if the encoding scheme includes a filter value), into a binary bit string that is suitable for storage in RFID Tags, including the EPC Memory Bank of a Gen 2 RFID Tag. Owing to tradeoffs between data capacity and the number of bits in the encoded value, more than one binary encoding scheme exists for certain EPC schemes.
EPC Binary Encoding Scheme	Section 13	A particular format for the encoding of an Electronic Product Code, together with a Filter Value in some cases, into an EPC Binary Encoding. Each EPC Scheme has at least one corresponding EPC Binary Encoding Scheme. from a specified combination of data elements. Owing to tradeoffs between data capacity and the number of bits in the encoded value, more than one binary encoding scheme exists for certain EPC schemes. An EPC Binary Encoding begins with an 8-bit header that identifies which binary encoding scheme is used for that binary encoding; this serves to identify how the remainder of the binary encoding is to be interpreted.
EPC Pure Identity URI	Section 6	See Pure Identity EPC URI.
EPC Raw URI	Section 12	A representation of the complete contents of the EPC Memory Bank of a Gen 2 RFID Tag,
EPC Scheme	Section 6	A particular format for the construction of an Electronic Product Code from a specified combination of data elements. A Pure Identity EPC URI begins with the name of the EPC Scheme used for that URI, which both serves to ensure global uniqueness of the complete URI as well as identify how the remainder of the URI is to be interpreted. Each type of GS1 Key has a corresponding EPC Scheme that allows for the construction of an EPC that corresponds to the value of a GS1 Key, under certain conditions. Other EPC Schemes exist that allow for construction of EPCs not related to GS1 keys.

Term	Defined Where	Meaning
EPC Tag URI	Section 12	A representation of the complete contents of the EPC Memory Bank of a Gen 2 RFID Tag, in the form of an Internet Uniform Resource Identifier that includes a decoded representation of EPC data fields, usable when the EPC Memory Bank contains a valid EPC Binary Encoding. Because the EPC Tag URI represents the complete contents of the EPC Memory Bank, it includes control information in addition to the EPC, in contrast to the Pure Identity EPC URI.
Extended Tag Identification (XTID)	Section 16	Information that may be included in the TID Bank of a Gen 2 RFID Tag in addition to the make and model information. The XTID may include a manufacturer-assigned unique serial number and may also include other information that describes the capabilities of the tag.
Filter Value	Section 10	A 3-bit field of control information that is stored in the EPC Memory Bank of a Gen 2 RFID Tag when the tag contains certain types of EPCs. The filter value makes it easier to read desired RFID Tags in an environment where there may be other tags present, such as reading a pallet tag in the presence of a large number of item-level tags.
Gen 2 RFID Tag	Section 8	An RFID Tag that conforms to one of the EPCglobal Gen 2 family of air interface protocols. This includes the UHF Class 1 Gen 2 Air Interface [UHFC1G2], and other standards currently under development within EPCglobal.
GS1 Company Prefix	[GS1GS10.0]	Part of the GS1 System identification number consisting of a GS1 Prefix and a Company Number, both of which are allocated by GS1 Member Organisations.
GS1 Element String	[GS1GS10.0]	The combination of a GS1 Application Identifier and GS1 Application Identifier Data Field.
GS1 Key	[GS1GS10.0]	A generic term for nine different identification keys defined in the GS1 General Specifications [GS1GS10.0], namely the GTIN, SSCC, GLN, GRAI, GIAI, GSRN, GDTI, GSIN, and GINC.

Term	Defined Where	Meaning
Pure Identity EPC URI	Section 6	The primary concrete representation of an Electronic Product Code. The Pure Identity EPC URI is an Internet Uniform Resource Identifier that contains an Electronic Product Code and no other information.
Radio-Frequency Identification (RFID) Tag		A data carrier that holds binary data, which may be affixed to a physical object, and which communicates the data to a interrogator ("reader") device through radio.
Reserved Bank (of a Gen 2 RFID Tag)	[UHFC1G2]	Bank 00 of a Gen 2 RFID Tag as specified in [UHFC1G2]. The Reserved Bank holds the access password and the kill password.
Tag Identification (TID)	[UHFC1G2]	Information that describes a Gen 2 RFID Tag itself, as opposed to describing the physical object to which the tag is affixed. The TID includes an indication of the make and model of the tag, and may also include Extended TID (XTID) information.
TID Bank (of a Gen 2 RFID Tag)	[UHFC1G2]	Bank 10 of a Gen 2 RFID Tag as specified in [UHFC1G2]. The TID Bank holds the TID and XTID (*q.v.*).
Uniform Resource Identifier (URI)	[RFC3986]	A compact sequence of characters that identifies an abstract or physical resource. A URI may be further classified as a Uniform Resource Name (URN) or a Uniform Resource Locator (URL), *q.v.*
Uniform Resource Locator (URL)	[RFC3986]	A Uniform Resource Identifier (URI) that, in addition to identifying a resource, provides a means of locating the resource by describing its primary access mechanism (e.g., its network "location").
Uniform Resource Name (URN)	[RFC3986], [RFC2141]	A Uniform Resource Identifier (URI) that is part of the `urn` scheme as specified by [RFC2141]. Such URIs refer to a specific resource independent of its network location or other method of access, or which may not have a network location at all. The term URN may also refer to any other URI having similar properties. Because an Electronic Product Code is a unique identifier for a physical object that does not necessarily have a network locatin or other method of access, URNs are used to represent EPCs.

Term	Defined Where	Meaning
User Memory Bank (of a Gen 2 RFID Tag)	[UHFC1G2]	Bank 11 of a Gen 2 RFID Tag as specified in [UHFC1G2]. The User Memory may be used to hold additional business data elements beyond the EPC.

Appendix C References

[ASN.1] CCITT, "Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)", CCITT Recommendation X.209, January 1988.

[EPCAF] F. Armenio et al, "EPCglobal Architecture Framework," EPCglobal Final Version 1.3, March 2009, http://www.epcglobalinc.org/standards/architecture/architecture_1_3-framework-20090319.pdf.

[GS1GS10.0] "GS1 General Specifications,- Version 10.0, Issue 1" January 2010, Published by GS1, Blue Tower, Avenue Louise 326, bte10, Brussels 1009, B-1050, Belgium, www.gs1.org.

[ISO15961] ISO/IEC, "Information technology – Radio frequency identification (RFID) for item management – Data protocol: application interface," ISO/IEC 15961:2004, October 2004.

[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. (When ISO/IEC 15962, 2nd Edition, is published, it should be used in prefrence to the earlier version. References herein to Annex D of [15962] refer only to ISO/IEC 15962, 2nd Edition or later.)

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

[RFC2141] R. Moats, "URN Syntax," RFC2141, May 1997, http://www.ietf.org/rfc/rfc2141.

[RFC3986] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC3986, January 2005, http://www.ietf.org/rfc/rfc3986.

[ONS1.0.1] EPCglobal, "EPCglobal Object Naming Service (ONS), Version 1.0.1," EPCglobal Ratified Standard, May 2008, http://www.epcglobalinc.org/standards/ons/ons_1_0_1-standard-20080529.pdf.

[UHFC1G2] EPCglobal, "EPC™ Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz – 960 MHz Version 1.2.0," EPCglobal Specification, May 2008, http://www.epcglobalinc.org/standards/uhfc1g2/uhfc1g2_1_2_0-standard-20080511.pdf.

[USDOD] "United States Department of Defense Suppliers' Passive RFID Information Guide," http://www.dodrfid.org/supplierguide.htm

Appendix D Extensible Bit Vectors

An Extensible Bit Vector (EBV) is a data structure with an extensible data range.

An EBV is an array of blocks. Each block contains a single extension bit followed by a specific number of data bits. If B is the total number of bits in one block, then a block contains B − 1 data bits. The notation EBV-*n* used in this specification indicates an EBV with a block size of *n*; e.g., EBV-8 denotes an EBV with B=8.

The data value represented by an EBV is simply the bit string formed by the data bits as read from left to right, ignoring all extension bits. The last block of an EBV has an extension bit of zero, and all blocks of an EBV preceding the last block (if any) have an extension bit of one.

The following table illustrates different values represented in EBV-6 format and EBV-8 format. Spaces are added to the EBVs for visual clarity.

Value	EBV-6	EBV-8
0	000000	00000000
1	000001	00000001
31 (2^5−1)	011111	00011111
32 (2^5)	100001 000000	00100000
33 (2^5+1)	100001 000001	00100001
127 (2^7−1)	100011 011111	01111111
128 (2^7)	100100 000000	10000001 00000000
129 (2^7+1)	100100 000001	10000001 00000001
16384 (2^{14})	110000 100000 000000	10000001 10000000 00000000

The Packed Objects specification in Appendix I makes use of EBV-3, EBV-6, and EBV-8.

Appendix E (non-normative) Examples: EPC Encoding and Decoding

This section presents two examples showing encoding and decoding between the Serialized Global Identification Number (SGTIN) and the EPC memory bank of a Gen 2 RFID tag.

As these are merely illustrative examples, in all cases the indicated normative sections of this specification should be consulted for the definitive rules for encoding and decoding. The diagrams and accompanying notes in this section are not intended to be a complete specification for encoding or decoding, but instead serve only to illustrate the highlights of how the normative encoding and decoding procedures function. The procedures for encoding other types of identifiers are different in significant ways, and the appropriate sections of this specification should be consulted.

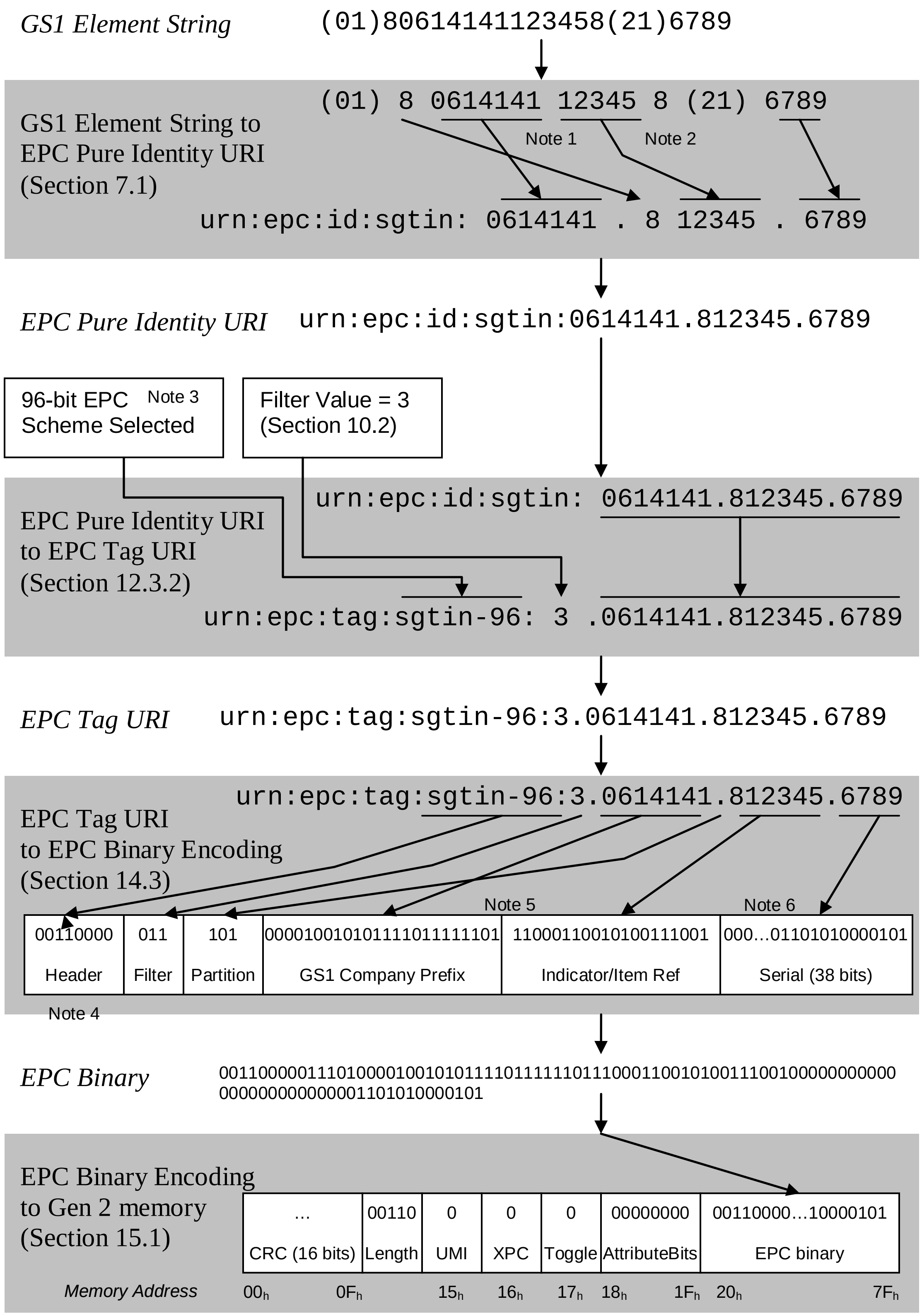
E.1 Encoding a Serialized Global Trade Item Number (SGTIN) to SGTIN-96

This example illustrates the encoding of a GS1 Element String containing a Serialized Global Trade Item Number (SGTIN) into an EPC Gen 2 RFID tag using the SGTIN-96 EPC scheme, with intermediate steps including the EPC URI, the EPC Tag URI, and the EPC Binary Encoding.

In some applications, only a part of this illustration is relevant. For example, an application may only need to transform a GS1 Element String into an EPC URI, in which case only the top of the illustration is needed.

The illustration below makes reference to the following notes:

- Note 1: The step of converting a GS1 Element String into the EPC Pure Identity URI requires that the number of digits in the GS1 Company Prefix be determined; e.g., by reference to an external table of company prefixes. In this example, the GS1 Company Prefix is shown to be seven digits.
- Note 2: The check digit in GTIN as it appears in the GS1 Element String is not included in the EPC Pure Identity URI.
- Note 3: The SGTIN-96 EPC scheme may only be used if the Serial Number meets certain constraints. Specifically, the serial number must (a) consist only of digit characters; (b) not begin with a zero digit (unless the entire serial number is the single digit '`0`'); and (c) correspond to a decimal numeral whose numeric value that is less than 2^{38} (less than 274,877,906,944). For all other serial numbers, the SGTIN-198 EPC scheme must be used. Note that the EPC URI is identical regardless of whether SGTIN-96 or SGTIN-198 is used in the RFID Tag.
- Note 4: EPC Binary Encoding header values are defined in Section 14.2.
- Note 5: The number of bits in the GS1 Company Prefix and Indicator/Item Reference fields in the EPC Binary Encoding depends on the number of digits in the GS1 Company Prefix portion of the EPC URI, and this is indicated by a code in the Partition field of the EPC Binary Encoding. See Table 17 (for the SGTIN EPC only).
- Note 6: The Serial field of the EPC Binary Encoding for SGTIN-96 is 38 bits; not all bits are shown here due to space limitations.



GS1 Element String
(01)80614141123458(21)6789
GS1 Element String to EPC Pure Identity URI (Section 7.1)
(01) 8 0614141 12345 8 (21) 6789
Note 1
Note 2
urn:epc:id:sgtin: 0614141 . 8 12345 . 6789
EPC Pure Identity URI
urn:epc:id:sgtin:0614141.812345.6789
96-bit EPC Scheme Selected
Note 3
Filter Value = 3 (Section 10.2)
EPC Pure Identity URI to EPC Tag URI (Section 12.3.2)
urn:epc:id:sgtin: 0614141.812345.6789
urn:epc:tag:sgtin-96: 3 .0614141.812345.6789
EPC Tag URI
urn:epc:tag:sgtin-96:3.0614141.812345.6789
EPC Tag URI to EPC Binary Encoding (Section 14.3)
urn:epc:tag:sgtin-96:3.0614141.812345.6789
Note 5
Note 6
00110000
Header
011
Filter
101
Partition
000010010101011110111111101
GS1 Company Prefix
11000110010100111001
Indicator/Item Ref
000…01101010000101
Serial (38 bits)
Note 4
EPC Binary
001100000111010000100101011110111111101110001100101001110010000000000
00000000000000001101010000101
EPC Binary Encoding to Gen 2 memory (Section 15.1)
…
CRC (16 bits)
00110
Length
0
UMI
0
XPC
0
Toggle
00000000
AttributeBits
00110000…10000101
EPC binary
Memory Address
00h
0Fh
15h
16h
17h
18h
1Fh
20h
7Fh

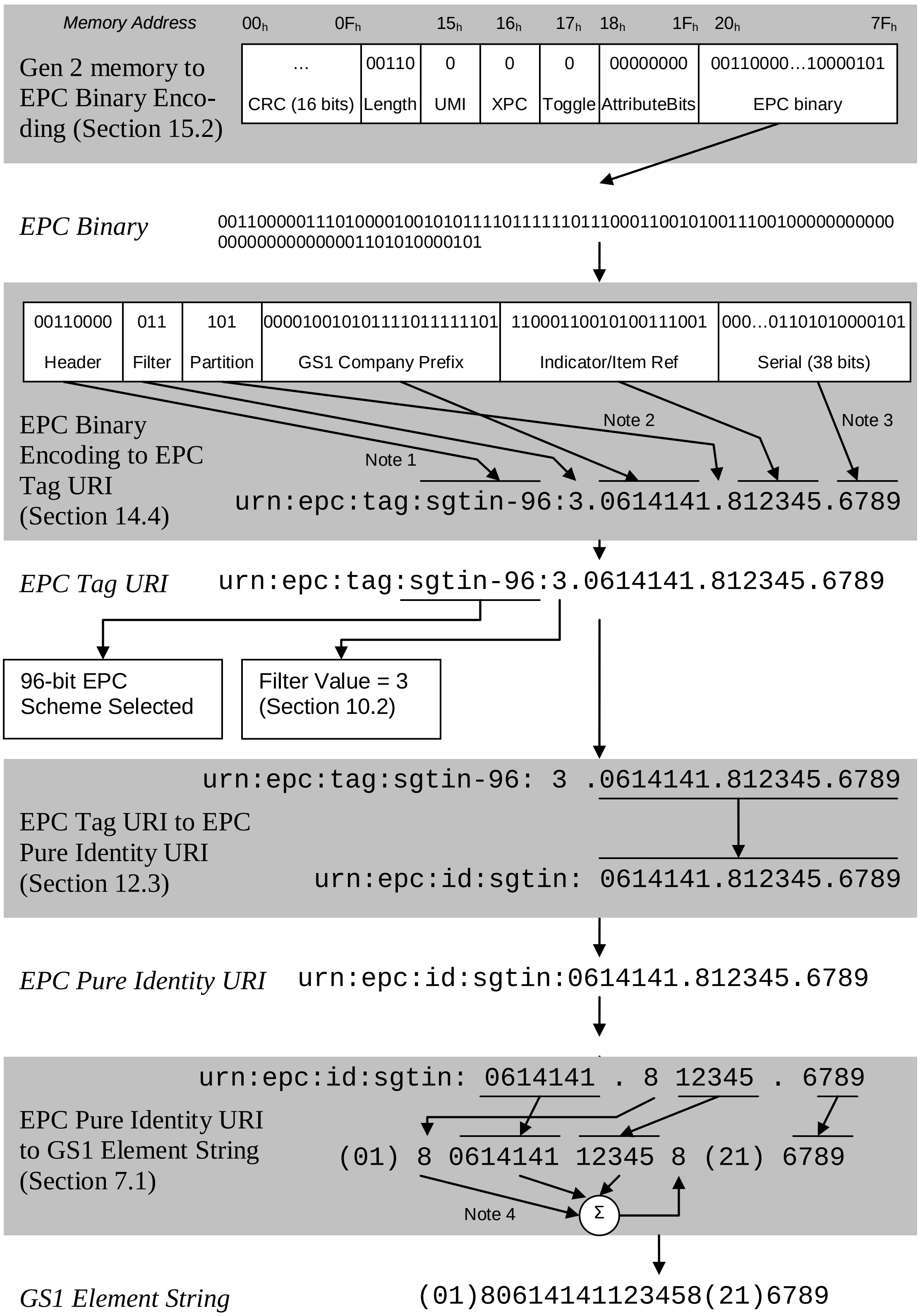
E.2 Decoding an SGTIN-96 to a Serialized Global Trade Item Number (SGTIN)

This example illustrates the decoding of an EPC Gen 2 RFID tag containing an SGTIN-96 EPC Binary Encoding into a GS1 Element String containing a Serialized Global Trade Item Number (SGTIN), with intermediate steps including the EPC Binary Encoding, the EPC Tag URI, and the EPC URI.

In some applications, only a part of this illustration is relevant. For example, an application may only need to convert an EPC URI to a GS1 Element String, in which case only the top of the illustration is needed.

The illustration below makes reference to the following notes:

- Note 1: The EPC Binary Encoding header indicates how to interpret the remainder of the binary data, and the EPC scheme name to be included in the EPC Tag URI. EPC Binary Encoding header values are defined in Section 14.2.
- Note 2: The Partition field of the EPC Binary Encoding contains a code that indicates the number of bits in the GS1 Company Prefix field and the Indicator/Item Reference field. The partition code also determines the number of decimal digits to be used for those fields in the EPC Tag URI (the decimal representation for those two fields is padded on the left with zero characters as necessary). See Table 17 (for the SGTIN EPC only).
- Note 3: For the SGTIN-96 EPC scheme, the Serial Number field is decoded by interpreting the bits as a binary integer and converting to a decimal numeral without leading zeros (unless all serial number bits are zero, which decodes as the string "`0`"). Serial numbers containing non-digit characters or that begin with leading zero characters may only be encoded in the SGTIN-198 EPC scheme.
- Note 4: The check digit in the GS1 Element String is calculated from other digits in the EPC Pure Identity URI, as specified in Section 7.1.



Memory Address
00h 0Fh 15h 16h 17h 18h 1Fh 20h 7Fh
Gen 2 memory to EPC Binary Encoding (Section 15.2)
… CRC (16 bits)
00110 Length
0 UMI
0 XPC
0 Toggle
00000000 AttributeBits
00110000…10000101 EPC binary
EPC Binary
0011000001110100001001010111101111111011100011001010011100100000000000
000000000000001101010000101
00110000 Header
011 Filter
101 Partition
000010010101111011111101 GS1 Company Prefix
11000110010100111001 Indicator/Item Ref
000…01101010000101 Serial (38 bits)
EPC Binary Encoding to EPC Tag URI (Section 14.4)
Note 1
Note 2
Note 3
urn:epc:tag:sgtin-96:3.0614141.812345.6789
EPC Tag URI
urn:epc:tag:sgtin-96:3.0614141.812345.6789
96-bit EPC Scheme Selected
Filter Value = 3 (Section 10.2)
urn:epc:tag:sgtin-96: 3 .0614141.812345.6789
EPC Tag URI to EPC Pure Identity URI (Section 12.3)
urn:epc:id:sgtin: 0614141.812345.6789
EPC Pure Identity URI
urn:epc:id:sgtin:0614141.812345.6789
urn:epc:id:sgtin: 0614141 . 8 12345 . 6789
EPC Pure Identity URI to GS1 Element String (Section 7.1)
(01) 8 0614141 12345 8 (21) 6789
Note 4
Σ
GS1 Element String
(01)80614141123458(21)6789

Appendix F Packed Objects ID Table for Data Format 9

This section provides the Packed Objects ID Table for Data Format 9, which defines Packed Objects ID values, OIDs, and format strings for GS1 Application Identifiers.

Section F.1 is a non-normative listing of the content of the ID Table for Data Format 9, in a human readable, tabular format. Section F.2 is the normative table, in machine readable, comma-separated-value format, as registered with ISO.

F.1 Tabular Format (non-normative)

This section is a non-normative listing of the content of the ID Table for Data Format 9, in a human readable, tabular format. See Section F.2 for the normative, machine readable, comma-separated-value format, as registered with ISO.

K-Text = GS1 AI ID Table for ISO/IEC 15961 Format 9						
K-Version = 1.00						
K-ISO15434=05						
K-Text = Primary Base Table						
K-TableID = F9B0						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 90						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
00	1	0	00	SSCC (Serial Shipping Container Code)	SSCC	18n
01	2	1	01	Global Trade Item Number	GTIN	14n
02 + 37	3	(2)(37)	(02)(37)	GTIN + Count of trade items contained in a logistic unit	CONTENT + COUNT	(14n)(1*8n)
10	4	10	10	Batch or lot number	BATCH/LOT	1*20an
11	5	11	11	Production date (YYMMDD)	PROD DATE	6n
12	6	12	12	Due date (YYMMDD)	DUE DATE	6n
13	7	13	13	Packaging date (YYMMDD)	PACK DATE	6n
15	8	15	15	Best before date (YYMMDD)	BEST BEFORE OR SELL BY	6n
17	9	17	17	Expiration date (YYMMDD)	USE BY OR EXPIRY	6n
20	10	20	20	Product variant	VARIANT	2n
21	11	21	21	Serial number	SERIAL	1*20an
22	12	22	22	Secondary data for specific health industry products	QTY/DATE/BATCH	1*29an
240	13	240	240	Additional product identification assigned by the manufacturer	ADDITIONAL ID	1*30an

241	14	241	241	Customer part number	CUST. PART NO.	1*30an
242	15	242	242	Made-to-Order Variation Number	VARIATION NUMBER	1*6n
250	16	250	250	Secondary serial number	SECONDARY SERIAL	1*30an
251	17	251	251	Reference to source entity	REF. TO SOURCE	1*30an
253	18	253	253	Global Document Type Identifier	DOC. ID	13*30n
30	19	30	30	Variable count	VAR. COUNT	1*8n
310n 320n etc	20	K-Secondary = S00		Net weight, kilograms or pounds or troy oz (Variable Measure Trade Item)		
311n 321n etc	21	K-Secondary = S01		Length of first dimension (Variable Measure Trade Item)		
312n 324n etc	22	K-Secondary = S02		Width, diameter, or second dimension (Variable Measure Trade Item)		
313n 327n etc	23	K-Secondary = S03		Depth, thickness, height, or third dimension (Variable Measure Trade Item)		
314n 350n etc	24	K-Secondary = S04		Area (Variable Measure Trade Item)		
315n 316n etc	25	K-Secondary = S05		Net volume (Variable Measure Trade Item)		
330n or 340n	26	330%x30-36 / 340%x30-36	330%x30-36 / 340%x30-36	Logistic weight, kilograms or pounds	GROSS WEIGHT (kg) or (lb)	6n / 6n
331n, 341n, etc	27	K-Secondary = S09		Length or first dimension		
332n, 344n, etc	28	K-Secondary = S10		Width, diameter, or second dimension		
333n, 347n, etc	29	K-Secondary = S11		Depth, thickness, height, or third dimension		
334n 353n etc	30	K-Secondary = S07		Logistic Area		
335n 336n etc	31	K-Secondary = S06	335%x30-36	Logistic volume		
337(***)	32	337%x30-36	337%x30-36	Kilograms per square metre	KG PER m^2	6n
390n or 391n	33	390%x30-39 / 391%x30-39	390%x30-39 / 391%x30-39	Amount payable – single monetary area or with ISO currency code	AMOUNT	1*15n / 4*18n

392n or 393n	34	392%x30-39 / 393%x30-39	392%x30-39 / 393%x30-39	Amount payable for Variable Measure Trade Item – single monetary unit or ISO cc	PRICE	1*15n / 4*18n
400	35	400	400	Customer's purchase order number	ORDER NUMBER	1*30an
401	36	401	401	Global Identification Number for Consignment	GINC	1*30an
402	37	402	402	Global Shipment Identification Number	GSIN	17n
403	38	403	403	Routing code	ROUTE	1*30an
410	39	410	410	Ship to - deliver to Global Location Number	SHIP TO LOC	13n
411	40	411	411	Bill to - invoice to Global Location Number	BILL TO	13n
412	41	412	412	Purchased from Global Location Number	PURCHASE FROM	13n
413	42	413	413	Ship for - deliver for - forward to Global Location Number	SHIP FOR LOC	13n
414 and 254	43	(414) [254]	(414) [254]	Identification of a physical location GLN, and optional Extension	LOC No + GLN EXTENSION	(13n) [1*20an]
415 and 8020	44	(415) (8020)	(415) (8020)	Global Location Number of the Invoicing Party and Payment Slip Reference Number	PAY + REF No	(13n) (1*25an)
420 or 421	45	(420/421)	(420/421)	Ship to - deliver to postal code	SHIP TO POST	(1*20an / 3n 1*9an)
422	46	422	422	Country of origin of a trade item	ORIGIN	3n
423	47	423	423	Country of initial processing	COUNTRY - INITIAL PROCESS.	3*15n
424	48	424	424	Country of processing	COUNTRY - PROCESS.	3n
425	49	425	425	Country of disassembly	COUNTRY - DISASSEMBLY	3n
426	50	426	426	Country covering full process chain	COUNTRY – FULL PROCESS	3n
7001	51	7001	7001	NATO stock number	NSN	13n
7002	52	7002	7002	UN/ECE meat carcasses and cuts classification	MEAT CUT	1*30an
7003	53	7003	7003	Expiration Date and Time	EXPIRY DATE/TIME	10n
7004	54	7004	7004	Active Potency	ACTIVE POTENCY	1*4n
703s	55	7030	7030	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
703s	56	7031	7031	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an

703s	57	7032	7032	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
703s	58	7033	7033	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
703s	59	7034	7034	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
703s	60	7035	7035	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
703s	61	7036	7036	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
703s	62	7037	7037	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
703s	63	7038	7038	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
703s	64	7039	7039	Approval number of processor with ISO country code	PROCESSOR # s	3n 1*27an
8001	65	8001	8001	Roll products - width, length, core diameter, direction, and splices	DIMENSIONS	14n
8002	66	8002	8002	Electronic serial identifier for cellular mobile telephones	CMT No	1*20an
8003	67	8003	8003	Global Returnable Asset Identifier	GRAI	14n 0*16an
8004	68	8004	8004	Global Individual Asset Identifier	GIAI	1*30an
8005	69	8005	8005	Price per unit of measure	PRICE PER UNIT	6n
8006	70	8006	8006	Identification of the component of a trade item	GCTIN	18n
8007	71	8007	8007	International Bank Account Number	IBAN	1*30an
8008	72	8008	8008	Date and time of production	PROD TIME	8*12n
8018	73	8018	8018	Global Service Relation Number	GSRN	18n
8100 8101 etc	74	K-Secondary = S08		Coupon Codes		
90	75	90	90	Information mutually agreed between trading partners (including FACT DIs)	INTERNAL	1*30an
91	76	91	91	Company internal information	INTERNAL	1*30an
92	77	92	92	Company internal information	INTERNAL	1*30an

93	78	93	93	Company internal information	INTERNAL	1*30an
94	79	94	94	Company internal information	INTERNAL	1*30an
95	80	95	95	Company internal information	INTERNAL	1*30an
96	81	96	96	Company internal information	INTERNAL	1*30an
97	82	97	97	Company internal information	INTERNAL	1*30an
98	83	98	98	Company internal information	INTERNAL	1*30an
99	84	99	99	Company internal information	INTERNAL	1*30an
K-TableEnd = F9B0						

3210

K-Text = Sec. IDT - Net weight, kilograms or pounds or troy oz (Variable Measure Trade Item)						
K-TableID = F9S00						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
310(***)	0	310%x30-36	310%x30-36	Net weight, kilograms (Variable Measure Trade Item)	NET WEIGHT (kg)	6n
320(***)	1	320%x30-36	320%x30-36	Net weight, pounds (Variable Measure Trade Item)	NET WEIGHT (lb)	6n
356(***)	2	356%x30-36	356%x30-36	Net weight, troy ounces (Variable Measure Trade Item)	NET WEIGHT (t)	6n
K-TableEnd = F9S00						

3211

K-Text = Sec. IDT - Length of first dimension (Variable Measure Trade Item)						
K-TableID = F9S01						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
311(***)	0	311%x30-36	311%x30-36	Length of first dimension, metres (Variable Measure Trade Item)	LENGTH (m)	6n
321(***)	1	321%x30-36	321%x30-36	Length or first dimension, inches (Variable Measure Trade Item)	LENGTH (i)	6n

322(***)	2	322%x30-36	322%x30-36	Length or first dimension, feet (Variable Measure Trade Item)	LENGTH (f)	6n
323(***)	3	323%x30-36	323%x30-36	Length or first dimension, yards (Variable Measure Trade Item)	LENGTH (y)	6n
K-TableEnd = F9S01						

3212

K-Text = Sec. IDT - Width, diameter, or second dimension (Variable Measure Trade Item)						
K-TableID = F9S02						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
312(***)	0	312%x30-36	312%x30-36	Width, diameter, or second dimension, metres (Variable Measure Trade Item)	WIDTH (m)	6n
324(***)	1	324%x30-36	324%x30-36	Width, diameter, or second dimension, inches (Variable Measure Trade Item)	WIDTH (i)	6n
325(***)	2	325%x30-36	325%x30-36	Width, diameter, or second dimension, (Variable Measure Trade Item)	WIDTH (f)	6n
326(***)	3	326%x30-36	326%x30-36	Width, diameter, or second dimension, yards (Variable Measure Trade Item)	WIDTH (y)	6n
K-TableEnd = F9S02						

3213

K-Text = Sec. IDT - Depth, thickness, height, or third dimension (Variable Measure Trade Item)						
K-TableID = F9S03						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
313(***)	0	313%x30-36	313%x30-36	Depth, thickness, height, or third dimension, metres (Variable Measure Trade Item)	HEIGHT (m)	6n
327(***)	1	327%x30-36	327%x30-36	Depth, thickness, height, or third dimension, inches (Variable Measure Trade Item)	HEIGHT (i)	6n

328(***)	2	328%x30-36	328%x30-36	Depth, thickness, height, or third dimension, feet (Variable Measure Trade Item)	HEIGHT (f)	6n
329(***)	3	329%x30-36	329%x30-36	Depth, thickness, height, or third dimension, yards (Variable Measure Trade Item)	HEIGHT (y)	6n
K-TableEnd = F9S03						

K-Text = Sec. IDT - Area (Variable Measure Trade Item)						
K-TableID = F9S04						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
314(***)	0	314%x30-36	314%x30-36	Area, square metres (Variable Measure Trade Item)	AREA (m2)	6n
350(***)	1	350%x30-36	350%x30-36	Area, square inches (Variable Measure Trade Item)	AREA (i2)	6n
351(***)	2	351%x30-36	351%x30-36	Area, square feet (Variable Measure Trade Item)	AREA (f2)	6n
352(***)	3	352%x30-36	352%x30-36	Area, square yards (Variable Measure Trade Item)	AREA (y2)	6n
K-TableEnd = F9S04						

K-Text = Sec. IDT - Net volume (Variable Measure Trade Item)						
K-TableID = F9S05						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 8						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
315(***)	0	315%x30-36	315%x30-36	Net volume, litres (Variable Measure Trade Item)	NET VOLUME (l)	6n
316(***)	1	316%x30-36	316%x30-36	Net volume, cubic metres (Variable Measure Trade Item)	NET VOLUME (m3)	6n
357(***)	2	357%x30-36	357%x30-36	Net weight (or volume), ounces (Variable Measure Trade Item)	NET VOLUME (oz)	6n

360(***)	3	360%x30-36	360%x30-36	Net volume, quarts (Variable Measure Trade Item)	NET VOLUME (q)	6n
361(***)	4	361%x30-36	361%x30-36	Net volume, gallons U.S. (Variable Measure Trade Item)	NET VOLUME (g)	6n
364(***)	5	364%x30-36	364%x30-36	Net volume, cubic inches	VOLUME (i3), log	6n
365(***)	6	365%x30-36	365%x30-36	Net volume, cubic feet (Variable Measure Trade Item)	VOLUME (f3), log	6n
366(***)	7	366%x30-36	366%x30-36	Net volume, cubic yards (Variable Measure Trade Item)	VOLUME (y3), log	6n
K-TableEnd = F9S05						

3216

K-Text = Sec. IDT - Logistic Volume						
K-TableID = F9S06						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 8						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
335(***)	0	335%x30-36	335%x30-36	Logistic volume, litres	VOLUME (l), log	6n
336(***)	1	336%x30-36	336%x30-36	Logistic volume, cubic meters	VOLUME (m3), log	6n
362(***)	2	362%x30-36	362%x30-36	Logistic volume, quarts	VOLUME (q), log	6n
363(***)	3	363%x30-36	363%x30-36	Logistic volume, gallons	VOLUME (g), log	6n
367(***)	4	367%x30-36	367%x30-36	Logistic volume, cubic inches	VOLUME (q), log	6n
368(***)	5	368%x30-36	368%x30-36	Logistic volume, cubic feet	VOLUME (g), log	6n
369(***)	6	369%x30-36	369%x30-36	Logistic volume, cubic yards	VOLUME (i3), log	6n
K-TableEnd = F9S06						

3217

K-Text = Sec. IDT - Logistic Area						
K-TableID = F9S07						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
334(***)	0	334%x30-36	334%x30-36	Area, square metres	AREA (m2), log	6n
353(***)	1	353%x30-36	353%x30-36	Area, square inches	AREA (i2), log	6n
354(***)	2	354%x30-36	354%x30-36	Area, square feet	AREA (f2), log	6n

355(***)	3	355%x30-36	355%x30-36	Area, square yards	AREA (y2), log	6n
K-TableEnd = F9S07						

K-Text = Sec. IDT - Coupon Codes						
K-TableID = F9S08						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 8						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
8100	0	8100	8100	GS1-128 Coupon Extended Code - NSC + Offer Code	-	6n
8101	1	8101	8101	GS1-128 Coupon Extended Code - NSC + Offer Code + end of offer code	-	10n
8102	2	8102	8102	GS1-128 Coupon Extended Code – NSC	-	2n
8110	3	8110	8110	Coupon Code Identification for Use in North America		1*30an
K-TableEnd = F9S08						

K-Text = Sec. IDT - Length or first dimension						
K-TableID = F9S09						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
331(***)	0	331%x30-36	331%x30-36	Length or first dimension, metres	LENGTH (m), log	6n
341(***)	1	341%x30-36	341%x30-36	Length or first dimension, inches	LENGTH (i), log	6n
342(***)	2	342%x30-36	342%x30-36	Length or first dimension, feet	LENGTH (f), log	6n
343(***)	3	343%x30-36	343%x30-36	Length or first dimension, yards	LENGTH (y), log	6n
K-TableEnd = F9S09						

K-Text = Sec. IDT - Width, diameter, or second dimension						
K-TableID = F9S10						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
332(***)	0	332%x30-36	332%x30-36	Width, diameter, or second dimension, metres	WIDTH (m), log	6n
344(***)	1	344%x30-36	344%x30-36	Width, diameter, or second dimension	WIDTH (i), log	6n
345(***)	2	345%x30-36	345%x30-36	Width, diameter, or second dimension	WIDTH (f), log	6n
346(***)	3	346%x30-36	346%x30-36	Width, diameter, or second dimension	WIDTH (y), log	6n
K-TableEnd = F9S10						

K-Text = Sec. IDT - Depth, thickness, height, or third dimension						
K-TableID = F9S11						
K-RootOID = urn:oid:1.0.15961.9						
K-IDsize = 4						
AI or AIs	IDvalue	OIDs	IDstring	Name	Data Title	FormatString
333(***)	0	333%x30-36	333%x30-36	Depth, thickness, height, or third dimension, metres	HEIGHT (m), log	6n
347(***)	1	347%x30-36	347%x30-36	Depth, thickness, height, or third dimension	HEIGHT (i), log	6n
348(***)	2	348%x30-36	348%x30-36	Depth, thickness, height, or third dimension	HEIGHT (f), log	6n
349(***)	3	349%x30-36	349%x30-36	Depth, thickness, height, or third dimension	HEIGHT (y), log	6n
K-TableEnd = F9S11						

F.2 Comma-Separated-Value (CSV) Format

This section is the Packed Objects ID Table for Data Format 9 (GS1 Application Identifiers) in machine readable, comma-separated-value format, as registered with ISO. See Section F.1 for a non-normative listing of the content of the ID Table for Data Format 9, in a human readable, tabular format.

In the comma-separated-value format, line breaks are significant. However, certain lines are too long to fit within the margins of this document. In the listing below, the symbol █ at the end of line indicates that the ID Table line is continued on the following line. Such a line shall be interpreted by concatenating the following line and omitting the █ symbol.

```
K-Text = GS1 AI ID Table for ISO/IEC 15961 Format 9,,,,,,
K-Version = 1.00,,,,,,
```

```
K-ISO15434=05,,,,,,
K-Text = Primary Base Table,,,,,,
K-TableID = F9B0,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 90,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
00,1,0,"00",SSCC (Serial Shipping Container Code),SSCC,18n
01,2,1,"01",Global Trade Item Number,GTIN,14n
02 + 37,3,(2)(37),(02)(37),GTIN + Count of trade items contained in a logistic unit,CONTENT + COUNT,(14n)(1*8n)
10,4,10,10,Batch or lot number,BATCH/LOT,1*20an
11,5,11,11,Production date (YYMMDD),PROD DATE,6n
12,6,12,12,Due date (YYMMDD),DUE DATE,6n
13,7,13,13,Packaging date (YYMMDD),PACK DATE,6n
15,8,15,15,Best before date (YYMMDD),BEST BEFORE OR SELL BY,6n
17,9,17,17,Expiration date (YYMMDD),USE BY OR EXPIRY,6n
20,10,20,20,Product variant,VARIANT,2n
21,11,21,21,Serial number,SERIAL,1*20an
22,12,22,22,Secondary data for specific health industry products ,QTY/DATE/BATCH,1*29an
240,13,240,240,Additional product identification assigned by the manufacturer,ADDITIONAL ID,1*30an
241,14,241,241,Customer part number,CUST. PART NO.,1*30an
242,15,242,242,Made-to-Order Variation Number,VARIATION NUMBER,1*6n
250,16,250,250,Secondary serial number,SECONDARY SERIAL,1*30an
251,17,251,251,Reference to source entity,REF. TO SOURCE ,1*30an
253,18,253,253,Global Document Type Identifier,DOC. ID,13*30n
30,19,30,30,Variable count,VAR. COUNT,1*8n
310n 320n etc,20,K-Secondary = S00,,"Net weight, kilograms or pounds or troy oz (Variable Measure Trade Item)",,
311n 321n etc,21,K-Secondary = S01,,Length of first dimension (Variable Measure Trade Item),,
312n 324n etc,22,K-Secondary = S02,,"Width, diameter, or second dimension (Variable Measure Trade Item)",,
313n 327n etc,23,K-Secondary = S03,,"Depth, thickness, height, or third dimension (Variable Measure Trade Item)",,
314n 350n etc,24,K-Secondary = S04,,Area (Variable Measure Trade Item),,
315n 316n etc,25,K-Secondary = S05,,Net volume (Variable Measure Trade Item),,
330n or 340n,26,330%x30-36 / 340%x30-36,330%x30-36 / 340%x30-36,"Logistic weight, kilograms or pounds",
GROSS WEIGHT (kg) or (lb),6n / 6n
"331n, 341n, etc",27,K-Secondary = S09,,Length or first dimension,,
"332n, 344n, etc",28,K-Secondary = S10,,"Width, diameter, or second dimension",,
"333n, 347n, etc",29,K-Secondary = S11,,"Depth, thickness, height, or third dimension",,
334n 353n etc,30,K-Secondary = S07,,Logistic Area,,
335n 336n etc,31,K-Secondary = S06,335%x30-36,Logistic volume,,
337(***),32,337%x30-36,337%x30-36,Kilograms per square metre,KG PER m²,6n
390n or 391n,33,390%x30-39 / 391%x30-39,390%x30-39 / 391%x30-39,Amount payable – single monetary area or with
ISO currency code,AMOUNT,1*15n / 4*18n
392n or 393n,34,392%x30-39 / 393%x30-39,392%x30-39 / 393%x30-39,Amount payable for Variable Measure Trade Item –
single monetary unit or ISO cc, PRICE,1*15n / 4*18n
400,35,400,400,Customer's purchase order number,ORDER NUMBER,1*30an
401,36,401,401,Global Identification Number for Consignment,GINC,1*30an
402,37,402,402,Global Shipment Identification Number,GSIN,17n
403,38,403,403,Routing code,ROUTE,1*30an
410,39,410,410,Ship to - deliver to Global Location Number ,SHIP TO LOC,13n
411,40,411,411,Bill to - invoice to Global Location Number,BILL TO ,13n
412,41,412,412,Purchased from Global Location Number,PURCHASE FROM,13n
413,42,413,413,Ship for - deliver for - forward to Global Location Number,SHIP FOR LOC,13n
414 and 254,43,(414) [254],(414) [254],"Identification of a physical location GLN, and optional Extension",LOC No +
GLN EXTENSION,(13n) [1*20an]
415 and 8020,44,(415) (8020),(415) (8020),Global Location Number of the Invoicing Party and Payment Slip Reference
Number,PAY + REF No,(13n) (1*25an)
420 or 421,45,(420/421),(420/421),Ship to - deliver to postal code,SHIP TO POST,(1*20an / 3n 1*9an)
422,46,422,422,Country of origin of a trade item,ORIGIN,3n
423,47,423,423,Country of initial processing,COUNTRY - INITIAL PROCESS.,3*15n
424,48,424,424,Country of processing,COUNTRY - PROCESS.,3n
425,49,425,425,Country of disassembly,COUNTRY - DISASSEMBLY,3n
426,50,426,426,Country covering full process chain,COUNTRY – FULL PROCESS,3n
7001,51,7001,7001,NATO stock number,NSN,13n
7002,52,7002,7002,UN/ECE meat carcasses and cuts classification,MEAT CUT,1*30an
7003,53,7003,7003,Expiration Date and Time,EXPIRY DATE/TIME,10n
7004,54,7004,7004,Active Potency,ACTIVE POTENCY,1*4n
703s,55,7030,7030,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,56,7031,7031,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,57,7032,7032,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,58,7033,7033,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,59,7034,7034,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,60,7035,7035,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,61,7036,7036,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,62,7037,7037,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,63,7038,7038,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
703s,64,7039,7039,Approval number of processor with ISO country code,PROCESSOR # s,3n  1*27an
8001,65,8001,8001,"Roll products - width, length, core diameter, direction, and splices",DIMENSIONS,14n
8002,66,8002,8002,Electronic serial identifier for cellular mobile telephones,CMT No,1*20an
8003,67,8003,8003,Global Returnable Asset Identifier,GRAI,14n 0*16an
8004,68,8004,8004,Global Individual Asset Identifier,GIAI,1*30an
8005,69,8005,8005,Price per unit of measure,PRICE PER UNIT,6n
8006,70,8006,8006,Identification of the component of a trade item,GCTIN,18n
8007,71,8007,8007,International Bank Account Number ,IBAN,1*30an
8008,72,8008,8008,Date and time of production,PROD TIME,8*12n
8018,73,8018,8018,Global Service Relation Number ,GSRN,18n
8100 8101 etc,74,K-Secondary = S08,,Coupon Codes,,
90,75,90,90,Information mutually agreed between trading partners (including FACT DIs),INTERNAL,1*30an
91,76,91,91,Company internal information,INTERNAL,1*30an
92,77,92,92,Company internal information,INTERNAL,1*30an
93,78,93,93,Company internal information,INTERNAL,1*30an
94,79,94,94,Company internal information,INTERNAL,1*30an
95,80,95,95,Company internal information,INTERNAL,1*30an
96,81,96,96,Company internal information,INTERNAL,1*30an
97,82,97,97,Company internal information,INTERNAL,1*30an
98,83,98,98,Company internal information,INTERNAL,1*30an
```



```
99,84,99,99,Company internal information,INTERNAL,1*30an

K-TableEnd = F9B0,,,,,,


"K-Text = Sec. IDT - Net weight, kilograms or pounds or troy oz (Variable Measure Trade Item)",,,,,,
K-TableID = F9S00,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
310(***),0,310%x30-36,310%x30-36,"Net weight, kilograms (Variable Measure Trade Item)",NET WEIGHT (kg),6n
320(***),1,320%x30-36,320%x30-36,"Net weight, pounds (Variable Measure Trade Item)",NET WEIGHT (lb),6n
356(***),2,356%x30-36,356%x30-36,"Net weight, troy ounces (Variable Measure Trade Item)",NET WEIGHT (t),6n
K-TableEnd = F9S00,,,,,,


K-Text = Sec. IDT - Length of first dimension (Variable Measure Trade Item),,,,,,
K-TableID = F9S01,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
311(***),0,311%x30-36,311%x30-36,"Length of first dimension, metres (Variable Measure Trade Item)",LENGTH (m),6n
321(***),1,321%x30-36,321%x30-36,"Length or first dimension, inches (Variable Measure Trade Item)",LENGTH (i),6n
322(***),2,322%x30-36,322%x30-36,"Length or first dimension, feet (Variable Measure Trade Item)",LENGTH (f),6n
323(***),3,323%x30-36,323%x30-36,"Length or first dimension, yards (Variable Measure Trade Item)",LENGTH (y),6n
K-TableEnd = F9S01,,,,,,


"K-Text = Sec. IDT - Width, diameter, or second dimension (Variable Measure Trade Item)",,,,,,
K-TableID = F9S02,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
312(***),0,312%x30-36,312%x30-36,"Width, diameter, or second dimension, metres (Variable Measure Trade Item)",
WIDTH (m),6n
324(***),1,324%x30-36,324%x30-36,"Width, diameter, or second dimension, inches (Variable Measure Trade Item)",
WIDTH (i),6n
325(***),2,325%x30-36,325%x30-36,"Width, diameter, or second dimension, (Variable Measure Trade Item)",
WIDTH (f),6n
326(***),3,326%x30-36,326%x30-36,"Width, diameter, or second dimension, yards (Variable Measure Trade Item)",
WIDTH (y),6n
K-TableEnd = F9S02,,,,,,


"K-Text = Sec. IDT - Depth, thickness, height, or third dimension (Variable Measure Trade Item)",,,,,,
K-TableID = F9S03,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
313(***),0,313%x30-36,313%x30-36,"Depth, thickness, height, or third dimension, metres (Variable Measure
Trade Item)",HEIGHT (m),6n
327(***),1,327%x30-36,327%x30-36,"Depth, thickness, height, or third dimension, inches (Variable Measure
Trade Item)",HEIGHT (i),6n
328(***),2,328%x30-36,328%x30-36,"Depth, thickness, height, or third dimension, feet (Variable Measure
Trade Item)",HEIGHT (f),6n
329(***),3,329%x30-36,329%x30-36,"Depth, thickness, height, or third dimension, yards (Variable Measure
Trade Item)",HEIGHT (y),6n
K-TableEnd = F9S03,,,,,,


K-Text = Sec. IDT - Area (Variable Measure Trade Item),,,,,,
K-TableID = F9S04,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
314(***),0,314%x30-36,314%x30-36,"Area, square metres (Variable Measure Trade Item)",AREA (m2),6n
350(***),1,350%x30-36,350%x30-36,"Area, square inches (Variable Measure Trade Item)",AREA (i2),6n
351(***),2,351%x30-36,351%x30-36,"Area, square feet (Variable Measure Trade Item)",AREA (f2),6n
352(***),3,352%x30-36,352%x30-36,"Area, square yards (Variable Measure Trade Item)",AREA (y2),6n
K-TableEnd = F9S04,,,,,,


K-Text = Sec. IDT - Net volume (Variable Measure Trade Item),,,,,,
K-TableID = F9S05,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 8,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
315(***),0,315%x30-36,315%x30-36,"Net volume, litres (Variable Measure Trade Item)",NET VOLUME (l),6n
316(***),1,316%x30-36,316%x30-36,"Net volume, cubic metres (Variable Measure Trade Item)",NET VOLUME (m3),6n
357(***),2,357%x30-36,357%x30-36,"Net weight (or volume), ounces (Variable Measure Trade Item)",NET VOLUME (oz),6n
360(***),3,360%x30-36,360%x30-36,"Net volume, quarts (Variable Measure  Trade Item)",NET VOLUME (q),6n
361(***),4,361%x30-36,361%x30-36,"Net volume, gallons U.S. (Variable Measure Trade Item)",NET VOLUME (g),6n
364(***),5,364%x30-36,364%x30-36,"Net volume, cubic inches","VOLUME (i3), log",6n
365(***),6,365%x30-36,365%x30-36,"Net volume, cubic feet (Variable Measure Trade Item)","VOLUME (f3), log",6n
366(***),7,366%x30-36,366%x30-36,"Net volume, cubic yards (Variable Measure Trade Item)","VOLUME (y3), log",6n
K-TableEnd = F9S05,,,,,,


K-Text = Sec. IDT - Logistic Volume,,,,,,
K-TableID = F9S06,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 8,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
335(***),0,335%x30-36,335%x30-36,"Logistic volume, litres","VOLUME (l), log",6n
```



```
336(***),1,336%x30-36,336%x30-36,"Logistic volume, cubic metres","VOLUME (m3), log",6n
362(***),2,362%x30-36,362%x30-36,"Logistic volume, quarts","VOLUME (q), log",6n
363(***),3,363%x30-36,363%x30-36,"Logistic volume, gallons","VOLUME (g), log",6n
367(***),4,367%x30-36,367%x30-36,"Logistic volume, cubic inches","VOLUME (q), log",6n
368(***),5,368%x30-36,368%x30-36,"Logistic volume, cubic feet","VOLUME (g), log",6n
369(***),6,369%x30-36,369%x30-36,"Logistic volume, cubic yards","VOLUME (i3), log",6n
K-TableEnd = F9S06,,,,,,


K-Text = Sec. IDT - Logistic Area,,,,,,
K-TableID = F9S07,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
334(***),0,334%x30-36,334%x30-36,"Area, square metres","AREA (m2), log",6n
353(***),1,353%x30-36,353%x30-36,"Area, square inches","AREA (i2), log",6n
354(***),2,354%x30-36,354%x30-36,"Area, square feet","AREA (f2), log",6n
355(***),3,355%x30-36,355%x30-36,"Area, square yards","AREA (y2), log",6n
K-TableEnd = F9S07,,,,,,


K-Text = Sec. IDT - Coupon Codes,,,,,,
K-TableID = F9S08,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 8,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
8100,0,8100,8100,GS1-128 Coupon Extended Code - NSC + Offer Code,-,6n
8101,1,8101,8101,GS1-128 Coupon Extended Code - NSC + Offer Code + end of offer code,-,10n
8102,2,8102,8102,GS1-128 Coupon Extended Code – NSC,-,2n
8110,3,8110,8110,Coupon Code Identification for Use in North America,,1*30an




K-TableEnd = F9S08,,,,,,


K-Text = Sec. IDT - Length or first dimension,,,,,,
K-TableID = F9S09,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
331(***),0,331%x30-36,331%x30-36,"Length or first dimension, metres","LENGTH (m), log",6n
341(***),1,341%x30-36,341%x30-36,"Length or first dimension, inches","LENGTH (i), log",6n
342(***),2,342%x30-36,342%x30-36,"Length or first dimension, feet","LENGTH (f), log",6n
343(***),3,343%x30-36,343%x30-36,"Length or first dimension, yards","LENGTH (y), log",6n
K-TableEnd = F9S09,,,,,,


"K-Text = Sec. IDT - Width, diameter, or second dimension",,,,,,
K-TableID = F9S10,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
332(***),0,332%x30-36,332%x30-36,"Width, diameter, or second dimension, metres","WIDTH (m), log",6n
344(***),1,344%x30-36,344%x30-36,"Width, diameter, or second dimension","WIDTH (i), log",6n
345(***),2,345%x30-36,345%x30-36,"Width, diameter, or second dimension","WIDTH (f), log",6n
346(***),3,346%x30-36,346%x30-36,"Width, diameter, or second dimension","WIDTH (y), log",6n
K-TableEnd = F9S10,,,,,,


"K-Text = Sec. IDT - Depth, thickness, height, or third dimension",,,,,,
K-TableID = F9S11,,,,,,
K-RootOID = urn:oid:1.0.15961.9,,,,,,
K-IDsize = 4,,,,,,
AI or AIs,IDvalue,OIDs,IDstring,Name,Data Title,FormatString
333(***),0,333%x30-36,333%x30-36,"Depth, thickness, height, or third dimension, metres","HEIGHT (m), log",6n
347(***),1,347%x30-36,347%x30-36,"Depth, thickness, height, or third dimension","HEIGHT (i), log",6n
348(***),2,348%x30-36,348%x30-36,"Depth, thickness, height, or third dimension","HEIGHT (f), log",6n
349(***),3,349%x30-36,349%x30-36,"Depth, thickness, height, or third dimension","HEIGHT (y), log",6n
K-TableEnd = F9S11,,,,,,
```

Appendix G (Intentionally Omitted)

[This appendix is omitted so that Appendices I through M, which specify packed objects, have the same appendix letters as the corresponding annexes of ISO/IEC 15962 , 2nd Edition.]

Appendix H (Intentionally Omitted)

[This appendix is omitted so that Appendices I through M, which specify packed objects, have the same appendix letters as the corresponding annexes of ISO/IEC 15962 , 2nd Edition.]

Appendix I Packed Objects Structure

I.1 Overview

The Packed Objects format provides for efficient encoding and access of user data. The Packed Objects format offers increased encoding efficiency compared to the No-Directory and Directory Access-Methods partly by utilizing sophisticated compaction methods, partly by defining an inherent directory structure at the front of each Packed Object (before any of its data is encoded) that supports random access while reducing the fixed overhead of some prior methods, and partly by utilizing data-system-specific information (such as the GS1 definitions of fixed-length Application Identifiers).

I.2 Overview of Packed Objects Documentation

The formal description of Packed Objects is presented in this Appendix and Appendices J, K, L, and M, as follows:

- The overall structure of Packed Objects is described in Section I.3.
- The individual sections of a Packed Object are described in Sections I.4 through I.9.
- The structure and features of ID Tables (utilized by Packed Objects to represent various data system identifiers) are described in Appendix J.
- The numerical bases and character sets used in Packed Objects are described in Appendix K.
- An encoding algorithm and worked example are described in Appendix L.
- The decoding algorithm for Packed Objects is described in Appendix M.

In addition, note that all descriptions of specific ID Tables for use with Packed Objects are registered separately, under the procedures of ISO/IEC 15961-2 as is the complete formal description of the machine-readable format for registered ID Tables.

I.3 High-Level Packed Objects Format Design

I.3.1 Overview

The Packed Objects memory format consists of a sequence in memory of one or more “Packed Objects” data structures. Each Packed Object may contain either encoded data or directory information, but not both. The first Packed Object in memory is preceded by a DSFID. The DSFID indicates use of Packed Objects as the memory’s Access Method, and indicates the registered Data Format that is the default format for every Packed Object in that memory. Every Packed Object may be optionally preceded or followed by padding patterns (if needed for alignment on word or block boundaries). In addition, at most one Packed Object in memory may optionally be preceded by a pointer to a Directory Packed Object (this pointer may itself be optionally followed by padding). This series of Packed Objects is terminated by optional padding followed by one or more zero-valued octets aligned on byte boundaries. See Figure I 3-1, which shows this sequence when appearing in an RFID tag.

NOTE: Because the data structures within an encoded Packed Object are bit-aligned rather than byte-aligned, this Appendix use the term ‘octet’ instead of ‘byte’ except in case where an eight-bit quantity must be aligned on a byte boundary.

Figure I 3-1: Overall Memory structure when using Packed Objects

DSFID	Optional Pointer* And/Or Padding	First Packed Object	Optional Pointer* And/Or Padding	Optional Second Packed Object	…	Optional Packed Object	Optional Pointer* And/Or Padding	Zero Octet(s)

*Note: the Optional Pointer to a Directory Packed Object may appear at most only once in memory

Every Packed Object represents a sequence of one or more data system Identifiers, each specified by reference to an entry within a Base ID Table from a registered data format. The entry is referenced by its relative position within the Base Table; this relative position or Base Table index is referred to throughout this specification as an “ID Value.” There are two different Packed Objects methods available for representing a sequence of Identifiers by reference to their ID Values:

- An ID List Packed Object (IDLPO) encodes a series of ID Values as a list, whose length depends on the number of data items being represented;
- An ID Map Packed Object (IDMPO) instead encodes a fixed-length bit array, whose length depends on the total number of entries defined in the registered Base Table. Each bit in the array is ‘1’ if the corresponding table entry is represented by the Packed Object, and is ‘0’ otherwise.

An ID List is the default Packed Objects format, because it uses fewer bits than an ID Map, if the list contains only a small percentage of the data system’s defined ID Values. However, if the Packed Object includes more than about one-quarter of the defined entries, then an ID Map requires fewer bits. For example, if a data system has sixteen entries, then each ID Value (table index) is a four bit quantity, and a list of four ID Values takes as many bits as would the complete ID Map. An ID Map’s fixed-length characteristic makes it especially suitable for use in a Directory Packed Object, which lists all of the Identifiers in all of the Packed Objects in memory (see section I.9). The overall structure of a Packed Object is the same, whether an IDLPO or an IDMPO, as shown in Figure I 3-2 and as described in the next subsection.

Figure I 3-2 Packed Object Structure

Optional Format Flags	Object Info Section (**IDLPO** or **IDMPO**)	Secondary ID Section (if needed)	Aux Format Section (if needed)	Data Section (if needed)

Packed Objects may be made “editable”, by adding an optional Addendum subsection to the end of the Object Info section, which includes a pointer to an “Addendum Packed Object” where additions and/or deletions have been made. One or more such “chains” of editable “parent” and “child” Packed Objects may be present within the overall sequence of Packed Objects in memory, but no more than one chain of Directory Packed Objects may be present.

I.3.2 Descriptions of each section of a Packed Object’s structure

Each Packed Object consists of several bit-aligned sections (that is, no pad bits between sections are used), carried in a variable number of octets. All required and optional Packed Objects formats are encompassed by the following ordered list of Packed Objects sections. Following this list, each Packed Objects section is introduced, and later sections of this Annex describe each Packed Objects section in detail.

- **Format Flags**: A Packed Object may optionally begin with the pattern ‘0000’ which is reserved to introduce one or more Format Flags, as described in I.4.2. These flags may indicate use of the non-default ID Map format. If the Format Flags are not present, then the Packed Object defaults to the ID List format.
    - Certain flag patterns indicate an inter-Object pattern (Directory Pointer or Padding)
    - Other flag patterns indicate the Packed Object’s type (Map or. List), and may indicated the presence of an optional Addendum subsection for editing.
- **Object Info:** All Packed Objects contain an Object Info Section which includes Object Length Information and ID Value Information:
    - Object Length Information includes an ObjectLength field (indicating the overall length of the Packed Object in octets) followed by Pad Indicator bit, so that the number of significant bits in the Packed Object can be determined.
    - ID Value Information indicates which Identifiers are present and in what order, and (if an IDLPO) also includes a leading NumberOfIDs field, indicating how many ID Values are encoded in the ID List.

  The Object Info section is encoded in one of the following formats, as shown in Figure I 3-3 and Figure I 3-4.

    - ID List (IDLPO) Object Info format:
        - Object Length (EBV-6) plus Pad Indicator bit
        - A single ID List or an ID Lists Section (depending on Format Flags)
    - ID Map (IDMPO) Object Info format:
        - One or more ID Map sections
        - Object Length (EBV-6) plus Pad Indicator bit

For either of these Object Info formats, an Optional Addendum subsection may be present at the end of the Object Info section.

- **Secondary ID Bits**: A Packed Object may include a Secondary ID section, if needed to encode additional bits that are defined for some classes of IDs (these bits complete the definition of the ID).
- **Aux Format Bits:** A Data Packed Object may include an Aux Format Section, which if present encodes one or more bits that are defined to support data compression, but do not contribute to defining the ID.
- **Data Section:** A Data Packed Object includes a Data Section, representing the compressed data associated with each of the identifiers listed within the Packed Object. This section is omitted in a Directory Packed Object, and in a Packed Object that uses No-directory compaction (see I.7.1). Depending on the declaration of data format in the relevant ID table, the Data section will contain either or both of two subsections:
  - **Known-Length Numerics subsection:** this subsection compacts and concatenates all of the non-empty data strings that are known a priori to be numeric.
  - **AlphaNumeric subsection:** this subsection concatenates and compacts all of the non-empty data strings that are not a priori known to be all-numeric.

Figure I 3-3: IDLPO Object Info Structure

Object Info, in a Default ID List PO			
Object Length	Number Of IDs	ID List	Optional Addendum

or

Object Info, in a Non-default ID List PO		
Object Length	ID Lists Section (one or more lists)	Optional Addendum

Figure I 3-4: IDMPO Object Info Structure

Object Info, in an ID Map PO		
ID Map Section (one or more maps)	Object Length	Optional Addendum

I.4 Format Flags section

The default layout of memory, under the Packed Objects access method, consists of a leading DSFID, immediately followed by an ID List Packed Object (at the next byte boundary), then optionally additional ID List Packed Objects (each beginning at the next byte boundary), and terminated by a zero-valued octet at the next byte boundary (indicating that no additional Packed Objects are encoded). This section defines the valid Format Flags patterns that may appear at the expected start of a Packed Object to override the default layout if desired (for example, by changing the Packed Object’s format, or by inserting padding patterns to align the next Packed Object on a word or block boundary). The set of defined patterns are shown in Table I 4-1.

Table I 4-1: Format Flags

Bit Pattern	Description	Additional Info	See Section
0000 0000	Termination Pattern	No more packed objects follow	I.4.1
LLLLLL xx	First octet of an IDLPO	For any LLLLLL > 3	I.5
0000	Format Flags starting pattern	(if the full EBV-6 is non-zero)	I.4.2
0000 10NA	IDLPO with: N = 1: non-default Info A = 1: Addendum Present	If N = 1: allows multiple ID tables If A = 1: Addendum ptr(s) at end of Object Info section	I.4.3
0000 01xx	Inter-PO pattern	A Directory Pointer, or padding	I.4.4
0000 0100	Signifies a padding octet	No padding length indicator follows	I.4.4
0000 0101	Signifies run-length padding	An EBV-8 padding length follows	I.4.4
0000 0110	RFU		I.4.4
0000 0111	Directory pointer	Followed by EBV-8 pattern	I.4.4
0000 11xx	ID Map Packed Object		I.4.2
0000 0001 0000 0010 0000 0011	[Invalid]	Invalid pattern	

I.4.1 Data Terminating Flag Pattern

A pattern of eight or more '0' bits at the expected start of a Packed Object denotes that no more Packed Objects are present in the remainder of memory.

NOTE: Six successive '0' bits at the expect start of a Packed Object would (if interpreted as a Packed Object) indicate an ID List Packed Object of length zero.

I.4.2 Format Flag section starting bit patterns

A non-zero EBV-6 with a leading pattern of "0000" is used as a Format Flags section Indication Pattern. The additional bits following an initial '0000' format Flag Indicating Pattern are defined as follows:

- A following two-bit pattern of '10' (creating an initial pattern of '000010') indicates an IDLPO with at least one non-default optional feature (see I.4.3)

- A following two-bit pattern of ‘11’ indicates an IDMPO, which is a Packed Object using an ID Map format instead of ID List-format The ID Map section (see I.9) immediately follows this two-bit pattern.
- A following two-bit pattern of ‘01’ signifies an External pattern (Padding pattern or Pointer) prior to the start of the next Packed Object (see I.4.4)

A leading EBV-6 Object Length of less than four is invalid as a Packed Objects length.

> NOTE: the shortest possible Packed Object is an IDLPO, for a data system using four bits per ID Value, encoding a single ID Value. This Packed Object has a total of 14 fixed bits. Therefore, a two-octet Packed Object would only contain two data bits, and is invalid. A three-octet Packed Object would be able to encode a single data item up to three digits long. In order to preserve “3” as an invalid length in this scenario, the Packed Objects encoder shall encode a leading Format Flags section (with all options set to zero, if desired) in order to increase the object length to four.

I.4.3 IDLPO Format Flags

The appearance of ‘000010’ at the expected start of a Packed Object is followed by two additional bits, to form a complete IDLPO Format Flags section of “000010NA”, where:

- If the first additional bit ‘N’ is ‘1’, then a non-default format is employed for the IDLPO Object Info section. Whereas the default IDLPO format allows for only a single ID List (utilizing the registration’s default Base ID Table), the optional non-default IDLPO Object Info format supports a sequence of one or more ID Lists, and each such list begins with identifying information as to which registered table it represents (see I.5.1).
- If the second additional bit ‘A’ is ‘1’, then an Addendum subsection is present at the end of the Object Info section (see I.5.6).

I.4.4 Patterns for use between Packed Objects

The appearance of ‘000001’ at the expected start of a Packed Object is used to indicate either padding or a directory pointer, as follows:

- A following two-bit pattern of ‘11’indicates that a Directory Packed Object Pointer follows the pattern. The pointer is one or more octets in length, in EBV-8 format. This pointer may be Null (a value of zero), but if non-zero, indicates the number of octets from the start of the pointer to the start of a Directory Packed Object (which if editable, shall be the first in its “chain”). For example, if the Format Flags byte for a Directory Pointer is encoded at byte offset 1, the Pointer itself occupies bytes beginning at offset 2, and the Directory starts at byte offset 9, then the Dir Ptr encodes the value “7” in EBV-8 format. A Directory Packed Object Pointer may appear before the first Packed Object in memory, or at any other position where a Packed Object may begin, but may only appear once in a given data carrier memory, and (if non-null) must be at a lower address than the Directory it points to. The first octet after this pointer may be padding (as defined immediately below), a new set of Format Flag patterns, or the start of an ID List Packed Object.

- A following two-bit pattern of '00' indicates that the full eight-bit pattern of '00000100' serves as a padding byte, so that the next Packed Object may begin on a desired word or block boundary. This pattern may repeat as necessary to achieve the desired alignment.
- A following two-bit pattern of '01' as a run-length padding indicator, and shall be immediately followed by an EBV-8 indicating the number of octets from the start of the EBV-8 itself to the start of the next Packed Object (for example, if the next Packed Object follows immediately, the EBV-8 has a value of one). This mechanism eliminates the need to write many words of memory in order to pad out a large memory block.
- A following two-bit pattern of '10' is Reserved.

I.5 Object Info Information

Each Packed Object's Object Info section contains both Length Information (the size of the Packed Object, in bits and in octets), and ID Values Information. A Packed Object encodes representations of one or more data system Identifiers and (if a Data Packed Object) also encodes their associated data elements (AI strings, DI strings, etc). The ID Values information encodes a complete listing of all the Identifiers (AIs, DIs, etc) encoded in the Packed Object, or (in a Directory Packed Object) all the Identifiers encoded anywhere in memory.

To conserve encoded and transmitted bits, data system Identifiers (each typically represented in data systems by either two, three, or four ASCII characters) is represented within a Packed Object by an ID Value, representing an index denoting an entry in a registered Base Table of ID Values. A single ID Value may represent a single Object Identifier, or may represent a commonly-used sequence of Object Identifiers. In some cases, the ID Value represents a "class" of related Object Identifiers, or an Object Identifier sequence in which one or more Object Identifiers are optionally encoded; in these cases, Secondary ID Bits (see I.6) are encoded in order to specify which selection or option was chosen when the Packed Object was encoded. A "fully-qualified ID Value" (FQIDV) is an ID Value, plus a particular choice of associated Secondary ID bits (if any are invoked by the ID Value's table entry). Only one instance of a particular fully-qualified ID Value may appear in a data carrier's Data Packed Objects, but a particular ID Value may appear more than once, if each time it is "qualified" by different Secondary ID Bits. If an ID Value does appear more than once, all occurrences shall be in a single Packed Object (or within a single "chain" of a Packed Object plus its Addenda).

There are two methods defined for encoding ID Values: an ID List Packed Object uses a variable-length list of ID Value bit fields, whereas an ID Map Packed Object uses a fixed-length bit array. Unless a Packed Object's format is modified by an initial Format Flags pattern, the Packed Object's format defaults to that of an ID List Packed Object (IDLPO), containing a single ID List, whose ID Values correspond to the default Base ID

Table of the registered Data Format. Optional Format Flags can change the format of the ID Section to either an IDMPO format, or to an IDLPO format encoding an ID Lists section (which supports multiple ID Tables, including non-default data systems).

Although the ordering of information within the Object Info section varies with the chosen format (see I.5.1), the Object Info section of every Packed Object shall provide Length information as defined in I.5.2, and ID Values information (see I.5.3) as defined in I.5.4, or I.5.5. The Object Info section (of either an IDLPO or an IDMPO) may conclude with an optional Addendum subsection (see I.5.6).

I.5.1 Object Info formats

I.5.1.1 IDLPO default Object Info format

The default IDLPO Object Info format is used for a Packed Object either without a leading Format Flags section, or with a Format Flags section indicating an IDLPO with a possible Addendum and a default Object Info section. The default IDLPO Object Info section contains a single ID List (optionally followed by an Addendum subsection if so indicated by the Format Flags). The format of the default IDLPO Object Info section is shown in Table I 5-1.

Table I 5-1: Default IDLPO Object Info format

Field Name:	**Length Information**	**NumberOfIDs**	**ID Listing**	**Addendum subsection**
Usage:	The number of octets in this Object, plus a last-octet pad indicator	number of ID Values in this Object (minus one)	A single list of ID Values; value size depends on registered Data Format	Optional pointer(s) to other Objects containing Edit information
Structure:	Variable: see I.5.2	Variable:EBV-3	See I.5.4	See I.5.6

In a IDLPO's Object Info section, the NumberOfIDs field is an EBV-3 Extensible Bit Vector, consisting of one or more repetitions of an Extension Bit followed by 2 value bits. This EBV-3 encodes one less than the number of ID Values on the associated ID Listing. For example, an EBV-3 of '101 000' indicates (4 + 0 +1) = 5 IDs values. The Length Information is as described in I.5.2 for all Packed Objects The next fields are an ID Listing (see I.5.4) and an optional Addendum subsection (see I.5.6).

I.5.1.2 IDLPO non-default Object Info format

Leading Format Flags may modify the Object Info structure of an IDLPO, so that it may contain more than one ID Listing, in an ID Lists section (which also allows non-default ID tables to be employed). The non-default IDLPO Object Info structure is shown in Table I 5-2.

Table I 5-2: Non-Default IDLPO Object Info format

Field Name:	**Length Info**	**ID Lists Section, first List**			**Optional Additional ID List(s)**	**Null App Indicator (single zero bit)**	**Addendum Subsection**
		Application Indicator	**Number of IDs**	**ID Listing**			
Usage:	The number of octets in this Object, plus a last-octet pad indicator	Indicates the selected ID Table and the size of each entry	Number Of ID Values on the list (minus one)	Listing of ID Values, then one F/R Use bit	Zero or more repeated lists, each for a different ID Table		Optional pointer(s) to other Objects containing Edit information
Structure:	see I.5.2	see I.5.3.1	See I.5.1.1	See I.5.4 and I .5.3.2	References in previous columns	See I.5.3.1	See I.5.6

I.5.1.3 IDMPO Object Info format

Leading Format Flags may define the Object Info structure to be an IDMPO, in which the Length Information (and optional Addendum subsection) follow an ID Map section (see I.5.5). This arrangement ensures that the ID Map is in a fixed location for a given application, of benefit when used as a Directory. The IDMPO Object Info structure is shown in Table I 5-3.

Table I 5-3: IDMPO Object Info format

Field Name:	**ID Map section**	**Length Information**	**Addendum**
Usage:	One or more ID Map structures, each using a different ID Table	The number of octets in this Object, plus a last-octet pad indicator	Optional pointer(s) to other Objects containing Edit information
Structure:	see I.9.1	See I.5.2	See I.5.6

I.5.2 Length Information

The format of the Length information, always present in the Object Info section of any Packed Object, is shown in table I 5-4.

Table I 5-4: Packed Object Length information

Field Name:	**ObjectLength**	**Pad Indicator**
Usage:	The number of 8-bit bytes in this Object This includes the 1st byte of this Packed Object, including its IDLPO/IDMPO format flags if present. It excludes patterns for use between packed objects, as specified in I.4.4	If ‘1’: the Object’s last byte contains at least 1 pad
Structure:	Variable: EBV-6	Fixed: 1 bit

The first field, ObjectLength, is an EBV-6 Extensible Bit Vector, consisting of one or more repetitions of an Extension Bit and 5 value bits. An EBV-6 of ‘000100’ (value of 4) indicates a four-byte Packed Object, An EBV-6 of ‘100001 000000’ (value of 32) indicates a 32-byte Object, and so on.

The Pad Indicator bit immediately follows the end of the EBV-6 ObjectLength. This bit is set to ‘0’ if there are no padding bits in the last byte of the Packed Object. If set to ‘1’, then bitwise padding begins with the least-significant or rightmost ‘1’ bit of the last byte, and the padding consists of this rightmost ‘1’ bit, plus any ‘0’ bits to the right of that bit. This method effectively uses a *single* bit to indicate a *three*-bit quantity (i.e., the number of trailing pad bits). When a receiving system wants to determine the total number of bits (rather than bytes) in a Packed Object, it would examine the ObjectLength field of the Packed Object (to determine the number of bytes) and multiply the result by eight, and (if the Pad Indicator bit is set) examine the last byte of the Packed Object and decrement the bit count by (1 plus the number of ‘0’ bits following the rightmost ‘1’ bit of that final byte).

I.5.3 General description of ID values

A registered data format defines (at a minimum) a Primary Base ID Table (a detailed specification for registered ID tables may be found in Annex J). This base table defines the data system Identifier(s) represented by each row of the table, any Secondary ID Bits or Aux Format bits invoked by each table entry, and various implicit rules (taken from a predefined rule set) that decoding systems shall use when interpreting data encoded according to each entry. When a data item is encoded in a Packed Object, its associated table entry is identified by the entry’s relative position in the Base Table. This table position or index is the ID Value that is represented in Packed Objects.

A Base Table containing a given number of entries inherently specifies the number of bits needed to encode a table index (i.e., an ID Value) in an ID List Packed Object (as the Log (base 2) of the number of entries). Since current and future data system ID Tables will vary in unpredictable ways in terms of their numbers of table entries, there is a need to pre-define an ID Value Size mechanism that allows for future extensibility to accommodate new tables, while minimizing decoder complexity and minimizing the need to upgrade decoding software (other than the addition of new tables). Therefore, regardless of the exact number of Base Table entries defined, each Base Table definition shall utilize one of the predefined sizes for ID Value encodings defined in Table I 5-5 (any unused entries shall be labeled as reserved, as provided in Annex J). The ID Size Bit pattern is encoded in a Packed Object only when it uses a non-default Base ID Table. Some entries in the table indicate a size that is not an integral power of two. When encoding (into an IDLPO) ID Values from tables that utilize such sizes, each pair of ID Values is encoded by multiplying the earlier ID of the pair by the base specified in the fourth column of Table I-5-5 and adding the later ID of the pair, and encoding the result in the number of bits specified in the fourth column. If there is a trailing single ID Value for this ID Table, it is encoded in the number of bits specified in the third column of Table I-5-5.

Table I 5-5: Defined ID Value sizes

ID Size Bit pattern	Maximum number of Table Entries	Number of Bits per single or trailing ID Value, and how encoded	Number of Bits per pair of ID Values, and how encoded
000	Up to 16	4, as 1 Base 16 value	8, as 2 Base 16 values
001	Up to 22	5, as 1 Base 22 value	9, as 2 Base 22 values
010	Up to 32	5, as 1 Base 32 value	10, as 2 Base 32 values
011	Up to 45	6, as 1 Base 45 value	11, as 2 Base 45 values
100	Up to 64	6, as 1 Base 64 value	12, as 2 Base 64 values
101	Up to 90	7, as 1 Base 90 value	13, as 2 Base 90 values
110	Up to 128	7, as 1 Base 128 value	14, as 2 Base 128 values
1110	Up to 256	8, as 1 Base 256 value	16, as 2 Base 256 values
111100	Up to 512	9, as 1 Base 512 value	18, as 2 Base 512 values
111101	Up to 1024	10, as 1 Base 1024 value	20, as 2 Base 1024 values
111110	Up to 2048	11, as 1 Base 2048 value	22, as 2 Base 2048 values
111111	Up to 4096	12, as 1 Base 4096 value	24, as 2 Base 4096 values

I.5.3.1 Application Indicator subsection

An Application Indicator subsection can be utilized to indicate use of ID Values from a default or non-default ID Table. This subsection is required in every IDMPO, but is only required in an IDLPO that uses the non-default format supporting multiple ID Lists.

An Application Indicator consists of the following components:

- A single AppIndicatorPresent bit, which if ‘0’ means that no additional ID List or Map follows. Note that this bit is always omitted for the first List or Map in an Object Info section. When this bit is present and ‘0’, then none of the following bit fields are encoded.

- A single ExternalReg bit that, if ‘1’, indicates use of an ID Table from a registration other than the memory’s default. If ‘1’, this bit is immediately followed by a 9-bit representation of a Data Format registered under ISO/IEC 15961.
- An ID Size pattern which denotes a table size (and therefore an ID Map bit length, when used in an IDMPO), which shall be one of the patterns defined by Table I 5-5. The table size indicated in this field must be less than or equal to the table size indicated in the selected ID table. The purpose of this field is so that the decoder can parse past the ID List or ID Map, even if the ID Table is not available to the decoder.
- a three-bit ID Subset pattern. The registered data format’s Primary Base ID Table, if used by the current Packed Object, shall always be indicated by an encoded ID Subset pattern of ‘000’. However, up to seven Alternate Base Tables may also be defined in the registration (with varying ID Sizes), and a choice from among these can be indicated by the encoded Subset pattern. This feature can be useful to define smaller sector-specific or application-specific subsets of a full data system, thus substantially reducing the size of the encoded ID Map.

I.5.3.2 Full/Restricted Use bits

When contemplating the use of new ID Table registrations, or registrations for external data systems, application designers may utilize a “restricted use” encoding option that adds some overhead to a Packed Object but in exchange results in a format that can be fully decoded by receiving systems not in possession of the new or external ID table. With the exception of a IDLPO using the default Object Info format, one Full/Restricted Use bit is encoded immediately after each ID table is represented in the ID Map section or ID Lists section of a Data or Directory Packed Object. In a Directory Packed object, this bit shall always be set to '0' and its value ignored. If an encoder wishes to utilize the “restricted use” option in an IDLPO, it shall preface the IDLPO with a Format Flags section invoking the non-default Object Info format.

If a “Full/Restricted Use” bit is ‘0’ then the encoding of data strings from the corresponding registered ID Table makes full use of the ID Table’s IDstring and FormatString information. If the bit is ‘1’, then this signifies that some encoding overhead was added to the Secondary ID section and (in the case of Packed-Object compaction) the Aux Format section, so that a decoder without access to the table can nonetheless output OIDs and data from the Packed Object according to the scheme specified in J.4.1. Specifically, a Full/Restricted Use bit set to ‘1’ indicates that:

- for each encoded ID Value, the encoder added an EBV-3 indicator to the Secondary ID section, to indicate how many Secondary ID bits were invoked by that ID Value. If the EBV-3 is nonzero, then the Secondary ID bits (as indicated by the table entry) immediately follow, followed in turn by another EBV-3, until the entire list of ID Values has been represented.
- the encoder did not take advantage of the information from the referenced table’s FormatString column. Instead, corresponding to each ID Value, the encoder inserted an EBV-3 into the Aux Format section, indicating the number of discrete data string lengths invoked by the ID Value (which could be more than one due to combinations

and/or optional components), followed by the indicated number of string lengths, each length encoded as though there were no FormatString in the ID table. All data items were encoded in the A/N subsection of the Data section.

I.5.4 ID Values representation in an ID Value-list Packed Object

Each ID Value is represented within an IDLPO on a list of bit fields; the number of bit fields on the list is determined from the NumberOfIDs field (see Table I 5-1). Each ID Value bit field's length is in the range of four to eleven bits, depending on the size of the Base Table index it represents. In the optional non-default format for an IDLPO's Object Info section, a single Packed Object may contain multiple ID List subsections, each referencing a different ID Table. In this non-default format, each ID List subsection consists of an Application Indicator subsection (which terminates the ID Lists, if it begins with a '0' bit), followed by an EBV-3 NumberOfIDs, an ID List, and a Full/Restricted Use flag.

I.5.5 ID Values representation in an ID Map Packed Object

Encoding an ID Map can be more efficient than encoding a list of ID Values, when representing a relatively large number of ID Values (constituting more than about 10 percent of a large Base Table's entries, or about 25 percent of a small Base Table's entries). When encoded in an ID Map, each ID Value is represented by its relative position within the map (for example, the first ID Map bit represents ID Value "0", the third bit represents ID Value "2", and the last bit represents ID Value 'n' (corresponding to the last entry of a Base Table with (n+1) entries). The value of each bit within an ID Map indicates whether the corresponding ID Value is present (if the bit is '1') or absent (if '0'). An ID Map is always encoded as part of an ID Map Section structure (see I.9.1).

I.5.6 Optional Addendum subsection of the Object Info section

The Packed Object Addendum feature supports basic editing operations, specifically the ability to add, delete, or replace individual data items in a previously-written Packed Object, without a need to rewrite the entire Packed Object. A Packed Object that does not contain an Addendum subsection cannot be edited in this fashion, and must be completely rewritten if changes are required.

An Addendum subsection consists of a Reverse Links bit, followed by a Child bit, followed by either one or two EBV-6 links. Links from a Data Packed Object shall only go to other Data Packed Objects as addenda; links from a Directory Packed Object shall only go to other Directory Packed Objects as addenda. The standard Packed Object structure rules apply, with some restrictions that are described in I.5.6.2.

The Reverse Links bit shall be set identically in every Packed Object of the same "chain." The Reverse Links bit is defined as follows:

- If the Reverse Links bit is '0', then each child in this chain of Packed Objects is at a higher memory location then its parent. The link to a Child is encoded as the number of octets (plus one) that are in between the last octet of the current Packed Object and the first octet of the Child. The link to the parent is encoded as the number of octets (plus one) that are in between the first octet of the parent Packed Object and the first octet of the current Packed Object.

- If the Reverse Links bit is ‘1’, then each child in this chain of Packed Objects is at a lower memory location then its parent. The link to a Child is encoded as the number of octets (plus one) that are in between the first octet of the current Packed Object and the first octet of the Child. The link to the parent is encoded as the number of octets (plus one) that are in between the last octet of the current Packed Object and the first octet of the parent.

The Child bit is defined as follows:

- If the Child bit is a ‘0’, then this Packed Object is an editable “Parentless” Packed Object (i.e., the first of a chain), and in this case the Child bit is immediately followed by a single EBV-6 link to the first “child” Packed Object that contains editing addenda for the parent.
- If the Child bit is a ‘1’, then this Packed Object is an editable “child” of an edited “parent,” and the bit is immediately followed by one EBV-6 link to the “parent” and a second EBV-6 line to the next “child” Packed Object that contains editing addenda for the parent.

A link value of zero is a Null pointer (no child exists), and in a Packed Object whose Child bit is ‘0’, this indicates that the Packed Object is editable, but has not yet been edited. A link to the Parent is provided, so that a Directory may indicate the presence and location of an ID Value in an Addendum Packed Object, while still providing an interrogator with the ability to efficiently locate the other ID Values that are logically associated with the original “parent” Packed Object. A link value of zero is invalid as a pointer towards a Parent.

In order to allow room for a sufficiently-large link, when the future location of the next “child” is unknown at the time the parent is encoded, it is permissible to use the “redundant” form of the EBV-6 (for example using “100000 000000” to represent a link value of zero).

I.5.6.1 Addendum “EditingOP” list (only in ID List Packed Objects)

In an IDLPO only, each Addendum section of a “child” ID List Packed Object contains a set of “EditingOp” bits encoded immediately after its last EBV-6 link. The number of such bits is determined from the number of entries on the Addendum Packed Object’s ID list. For each ID Value on this list, the corresponding EditingOp bit or bits are defined as follows:

- ‘1’ means that the corresponding Fully-Qualified ID Value (FQIDV) is Replaced. A Replace operation has the effect that the data originally associated with the FQIDV matching the FQIDV in this Addendum Packed Object shall be ignored, and logically replaced by the Aux Format bits and data encoded in this Addendum Packed Object)
- ‘00’ means that the corresponding FQIDV is Deleted but not replaced. In this case, neither the Aux Format bits nor the data associated with this ID Value are encoded in the Addendum Packed Object.

- '01' means that the corresponding FQIDV is Added (either this FQIDV was not previously encoded, or it was previously deleted without replacement). In this case, the associated Aux Format Bits and data shall be encoded in the Addendum Packed Object.

  NOTE: if an application requests several "edit" operations at once (including some Delete or Replace operations as well as Adds) then implementations can achieve more efficient encoding if the Adds share the Addendum overhead, rather than being implemented in a new Packed Object.

I.5.6.2 Packed Objects containing an Addendum subsection

A Packed Object containing an Addendum subsection is otherwise identical in structure to other Packed Objects. However, the following observations apply:

- A "parentless" Packed Object (the first in a chain) may be either an ID List Packed Object or an ID Map Packed Object (and a parentless IDMPO may be either a Data or Directory IDMPO). When a "parentless" PO is a directory, only directory IDMPOs may be used as addenda. A Directory IDMPO's Map bits shall be updated to correctly reflect the end state of the chain of additions and deletions to the memory bank; an Addendum to the Directory is not utilized to perform this maintenance (a Directory Addendum may only add new structural components, as described later in this section). In contrast, when the edited parentless object is an ID List Packed Object or ID Map Packed Object, its ID List or ID Map cannot be updated to reflect the end state of the aggregate Object (parents plus children).
- Although a "child" may be either an ID List or an ID Map Packed Object, only an IDLPO can indicate deletions or changes to the current set of fully-qualified ID Values and associated data that is embodied in the chain.
  - When a child is an IDMPO, it shall only be utilized to add (not delete or modify) structural information, and shall not be used to modify existing information. In a Directory chain, a child IDMPO may add new ID tables, or may add a new AuxMap section or subsections, or may extend an existing PO Index Table or ObjectOffsets list. In a Data chain, an IDMPO shall not be used as an Addendum, except to add new ID Tables.
  - When a child is an IDLPO, its ID list (followed by "EditingOp" bits) lists only those FQIDVs that have been deleted, added, or replaced, relative to the cumulative ID list from the prior Objects linked to it.

I.6 Secondary ID Bits section

The Packed Objects design requirements include a requirement that all of the data system Identifiers (AI's, DI's, etc.) encoded in a Packed Object's can be fully recognized without expanding the compressed data, even though some ID Values provide only a partially-qualified Identifier. As a result, if any of the ID Values invoke Secondary ID bits, the

Object Info section shall be followed by a Secondary ID Bits section. Examples include a four-bit field to identify the third digit of a group of related Logistics AIs.

Secondary ID bits can be invoked for several reasons, as needed in order to fully specify Identifiers. For example, a single ID Table entry's ID Value may specify a choice between two similar identifiers (requiring one encoded bit to select one of the two IDs at the time of encoding), or may specify a combination of required and optional identifiers (requiring one encoded bit to enable or disable each option). The available mechanisms are described in Annex J. All resulting Secondary ID bit fields are concatenated in this Secondary ID Bits section, in the same order as the ID Values that invoked them were listed within the Packed Object. Note that the Secondary ID Bits section is identically defined, whether the Packed Object is an IDLPO or an IDMPO, but is not present in a Directory IDMPO.

I.7 Aux Format section

The Aux Format section of a Data Packed Object encodes auxiliary information for the decoding process. A Directory Packed Object does not contain an Aux Format section. In a Data Packed Object, the Aux Format section begins with "Compact-Parameter" bits as defined in Table I.7-1.

Table I.7-1: Compact-Parameter bit patterns

Bit Pattern	Compaction method used in this Packed Object	Reference
'1'	"Packed-Object" compaction	See I.7.2
'000'	"Application-Defined", as defined for the No-Directory access method	See I.7.1
'001'	"Compact", as defined for the No-Directory access method	See I.7.1
'010'	"UTF-8", as defined for the No-Directory access method	See I.7.1
'011bbbb'	('bbbb' shall be in the range of 4..14): reserved for future definition	See I.7.1

If the Compact-Parameter bit pattern is '1', then the remainder of the Aux Format section is encoded as described in I.7.2; otherwise, the remainder of the Aux Format section is encoded as described in I.7.1.

I.7.1 Support for No-Directory compaction methods

If any of the No-Directory compaction methods were selected by the Compact-Parameter bits, then the Compact-Parameter bits are followed by an byte-alignment padding pattern consisting of zero or more '0' bits followed by a single '1' bit, so that the next bit after the '1' is aligned as the most-significant bit of the next byte.

This next byte is defined as the first octet of a "No-Directory Data section", which is used in place of the Data section described in I.8. The data strings of this Packed Object are encoded in the order indicated by the Object Info section of the Packed Object, compacted exactly as described in Annex D of [ISO15962] (Encoding rules for No-Directory Access-Method), with the following two exceptions:

- The Object-Identifier is not encoded in the “No-Directory Data section”, because it has already been encoded into the Object Info and Secondary ID sections.
- The Precursor is modified in that only the three Compaction Type Code bits are significant, and the other bits in the Precursor are set to ‘0’.

Therefore, each of the data strings invoked by the ID Table entry are separately encoded in a modified data set structure as:

<modified precursor> <length of compacted object> <compacted object octets>

The <compacted object octets> are determined and encoded as described in D.1.1 and D.1.2 of [ISO15962] and the <length of compacted object> is determined and encoded as described in D.2 of [ISO15962].

Following the last data set, a terminating precursor value of zero shall not be encoded (the decoding system recognizes the end of the data using the encoded ObjectLength of the Packed Object).

I.7.2 Support for the Packed-Object compaction method

If the Packed-Object compaction method was selected by the Compact-Parameter bits, then the Compact-Parameter bits are followed by zero or more Aux Format bits, as may be invoked by the ID Table entries used in this Packed Object. The Aux Format bits are then immediately followed by a Data section that uses the Packed-Object compaction method described in I.8.

An ID Table entry that was designed for use with the Packed-Object compaction method can call for various types of auxiliary information beyond the complete indication of the ID itself (such as bit fields to indicate a variable data length, to aid the data compaction process). All such bit fields are concatenated in this portion, in the order called for by the ID List or Map. Note that the Aux Format section is identically defined, whether the Packed Object is an IDLPO or an IDMPO.

An ID Table entry invokes Aux Format length bits for all entries that are not specified as fixed-length in the table (however, these length bits are not actually encoded if they correspond to the last data item encoded in the A/N subsection of a Packed Object). This information allows the decoding system to parse the decoded data into strings of the appropriate lengths. An encoded Aux Format length entry utilizes a variable number of bits, determined from the specified range between the shortest and longest data strings allowed for the data item, as follows:

- If a maximum length is specified, and the specified range (defined as the maximum length minus the minimum length) is less than eight, or greater than 44, then lengths in this range are encoded in the fewest number of bits that can express lengths within that range, and an encoded value of zero represents the minimum length specified in the format string. For example, if the range is specified as from three to six characters, then lengths are encoded using two bits, and ‘00’ represents a length of three.

- Otherwise (including the case of an unspecified maximum length), the value (actual length – specified minimum) is encoded in a variable number of bits, as follows:
  - Values from 0 to 14 (representing lengths from 1 to 15, if the specified minimum length is one character, for example) are encoded in four bits
  - Values from 15 to 29 are encoded in eight bits (a prefix of ‘1111’ followed by four bits representing values from 15 (‘0000’) to 29 (‘1110’)
  - Values from 30 to 44 are encoded in twelve bits (a prefix of ‘1111 1111’ followed by four bits representing values from 30 (‘0000’) to 44 (‘1110’)
  - Values greater than 44 are encoded as a twelve-bit prefix of all ‘1’s, followed by an EBV-6 indication of (value – 44).
- Notes:
  - if a range is specified with identical upper and lower bounds (i.e., a range of zero), this is treated as a fixed length, not a variable length, and no Aux Format bits are invoked.
  - If a range is unspecified, or has unspecified upper or lower bounds, then this is treated as a default lower bound of one, and/or an unlimited upper bound.

I.8 Data section

A Data section is always present in a Packed Object, except in the case of a Directory Packed Object or Directory Addendum Packed Object (which encode no data elements), the case of a Data Addendum Packed Object containing only Delete operations, and the case of a Packed Object that uses No-directory compaction (see I.7.1). When a Data section is present, it follows the Object Info section (and the Secondary ID and Aux Format sections, if present). Depending on the characteristics of the encoded IDs and data strings, the Data section may include one or both of two subsections in the following order: a Known-Length Numerics subsection, and an AlphaNumerics subsection. The following paragraphs provide detailed descriptions of each of these Data Section subsections. If all of the subsections of the Data section are utilized in a Packed Object, then the layout of the Data section is as shown in Figure I 8-1.

Figure I 8-1: Maximum Structure of a Packed Objects Data section

Known-Length Numeric subsection				AlphaNumeric subsection							
				A/N Header Bits				Binary Data Segments			
1st KLN Binary	2nd KLN Binary	…	Last KLN Binary	Non-Num Base Bit(s)	Prefix Bit, Prefix Run(s)	Suffix Bit, Suffix Run(s)	Char Map	Ext’d. Num Binary	Ext’d Non-Num Binary	Base 10 Binary	Non-Num Binary

I.8.1 Known-length-Numerics subsection of the Data Section

For always-numeric data strings, the ID table may indicate a fixed number of digits (this fixed-length information is not encoded in the Packed Object) and/or a variable number of digits (in which case the string's length was encoded in the Aux Format section, as described above). When a single data item is specified in the FormatString column (see J.2.3) as containing a fixed-length numeric string followed by a variable-length string, the numeric string is encoded in the Known-length-numerics subsection and the alphanumeric string in the Alphanumeric subsection.

The summation of fixed-length information (derived directly from the ID table) plus variable-length information (derived from encoded bits as just described) results in a "known-length entry" for each of the always-numeric strings encoded in the current Packed Object. Each all-numeric data string in a Packed Object (if described as all-numeric in the ID Table) is encoded by converting the digit string into a single Binary number (up to 160 bits, representing a binary value between 0 and (10^{48}-1)). Figure K-1 in Annex K shows the number of bits required to represent a given number of digits. If an all-numeric string contains more than 48 digits, then the first 48 are encoded as one 160-bit group, followed by the next group of up to 48 digits, and so on. Finally, the Binary values for each all-numeric data string in the Object are themselves concatenated to form the Known-length-Numerics subsection.

I.8.2 Alphanumeric subsection of the Data section

The Alphanumeric (A/N) subsection, if present, encodes all of the Packed Object's data from any data strings that were not already encoded in the Known-length Numerics subsection. If there are no alphanumeric characters to encode, the entire A/N subsection is omitted. The Alphanumeric subsection can encode any mix of digits and non-digit ASCII characters, or eight-bit data. The digit characters within this data are encoded separately, at an average efficiency of 4.322 bits per digit or better, depending on the character sequence. The non-digit characters are independently encoded at an average efficiency that varies between 5.91 bits per character or better (all uppercase letters), to a worst-case limit of 9 bits per character (if the character mix requires Base 256 encoding of non-numeric characters).

An Alphanumeric subsection consists of a series of A/N Header bits (see I.8.2.1), followed by from one to four Binary segments (each segment representing data encoded in a single numerical Base, such as Base 10 or Base 30, see I.8.2.4), padded if necessary to complete the final byte (see I 8.2.5).

I.8.2.1 A/N Header Bits

The A/N Header Bits are defined as follows:

- One or two Non-Numeric Base bits, as follows:
    - '0' indicates that Base 30 was chosen for the non-numeric Base;

    - '10' indicates that Base 74 was chosen for the non-numeric Base;
    - '11' indicates that Base 256 was chosen for the non-numeric Base
- Either a single '0' bit (indicating that no Character Map Prefix is encoded), or a '1' bit followed by one or more "Runs" of six Prefix bits as defined in I.8.2.3.
- Either a single '0' bit (indicating that no Character Map Suffix is encoded), or a '1' bit followed by one or more "Runs" of six Suffix bits as defined in I.8.2.3.
- A variable-length "Character Map" bit pattern (see I.8.2.2), representing the base of each of the data characters, if any, that were not accounted for by a Prefix or Suffix.

I.8.2.2 Dual-base Character-map encoding

Compaction of the ordered list of alphanumeric data strings (excluding those data strings already encoded in the Known-Length Numerics subsection) is achieved by first concatenating the data characters into a single data string (the individual string lengths have already been recorded in the Aux Format section). Each of the data characters is classified as either Base 10 (for numeric digits), Base 30 non-numerics (primarily uppercase A-Z), Base 74 non-numerics (which includes both uppercase and lowercase alphas, and other ASCII characters), or Base 256 characters. These character sets are fully defined in Annex K. All characters from the Base 74 set are also accessible from Base 30 via the use of an extra "shift" value (as are most of the lower 128 characters in the Base 256 set). Depending on the relative percentage of "native" Base 30 values vs. other values in the data string, one of those bases is selected as the more efficient choice for a non-numeric base.

Next, the precise sequence of numeric and non-numeric characters is recorded and encoded, using a variable-length bit pattern, called a "character map," where each '0' represents a Base 10 value (encoding a digit) and each '1' represents a value for a non-numeric character (in the selected base). Note that, (for example) if Base 30 encoding was selected, each data character (other than uppercase letters and the space character) needs to be represented by a pair of base-30 values, and thus each such data character is represented by a *pair* of '1' bits in the character map.

I.8.2.3 Prefix and Suffix Run-Length encoding

For improved efficiency in cases where the concatenated sequence includes runs of six or more values from the same base, provision is made for optional run-length representations of one or more Prefix or Suffix "Runs" (single-base character sequences), which can replace the first and/or last portions of the character map. The encoder shall not create a Run that separates a Shift value from its next (shifted) value, and thus a Run always represents an integral number of source characters.

An optional Prefix Representation, if present, consists of one or more occurrences of a Prefix Run. Each Prefix Run consists of one Run Position bit, followed by two Basis Bits, then followed by three Run Length bits, defined as follows:

- The Run Position bit, if '0', indicates that at least one more Prefix Run is encoded following this one (representing another set of source characters to the right of the current set). The Run Position bit, if ‘1’, indicates that the current Prefix Run is the last (rightmost) Prefix Run of the A/N subsection.

- The first basis bit indicates a choice of numeric vs. non-numeric base, and the second basis bit, if ‘1’, indicates that the chosen base is extended to include characters from the “opposite” base. Thus, ‘00’ indicates a run-length-encoded sequence of base 10 values; ‘01’ indicates a sequence that is primarily (but not entirely) digits, encoded in Base 13; ‘10’ indicates a sequence a sequence of values from the non-numeric base that was selected earlier in the A/N header, and ‘11’ indicates a sequence of values primarily from that non-numeric base, but extended to include digit characters as well. Note an exception: if the non-numeric base that was selected in the A/N header is Base 256, then the “extended” version is defined to be Base 40.
- The 3-bit Run Length value assumes a minimum useable run of six same-base characters, and the length value is further divided by 2. Thus, the possible 3-bit Run Length values of 0, 1, 2, … 7 indicate a Run of 6, 8, 10, … 20 characters from the same base. Note that a trailing “odd” character value at the end of a same-base sequence must be represented by adding a bit to the Character Map.

An optional Suffix Representation, if present, is a series of one or more Suffix Runs, each identical in format to the Prefix Run just described. Consistent with that description, note that the Run Position bit, if ‘1’, indicates that the current Suffix Run is the last (rightmost) Suffix Run of the A/N subsection, and thus any preceding Suffix Runs represented source characters to the left of this final Suffix Run.

I.8.2.4 Encoding into Binary Segments

Immediately after the last bit of the Character Map, up to four binary numbers are encoded, each representing all of the characters that were encoded in a single base system. First, a base-13 bit sequence is encoded (if one or more Prefix or Suffix Runs called for base-13 encoding). If present, this bit sequence directly represents the binary number resulting from encoding the combined sequence of all Prefix and Suffix characters (in that order) classified as Base 13 (ignoring any intervening characters not thus classified) as a single value, or in other words, applying a base 13 to Binary conversion. The number of bits to encode in this sequence is directly determined from the number of base-13 values being represented, as called for by the sum of the Prefix and Suffix Run lengths for base 13 sequences. The number of bits, for a given number of Base 13 values, is determined from the Figure in Annex K. Next, an Extended-NonNumeric Base segment (either Base-40 or Base 84) is similarly encoded (if any Prefix or Suffix Runs called for Extended-NonNumeric encoding).

Next, a Base-10 Binary segment is encoded that directly represents the binary number resulting from encoding the sequence of the digits in the Prefix and/or character map and/or Suffix (ignoring any intervening non-digit characters) as a single value, or in other words, applying a base 10 to Binary conversion. The number of bits to encode in this sequence is directly determined from the number of digits being represented, as shown in Annex K.

Immediately after the last bit of the Base-10 bit sequence (if any), a non-numeric (Base 30, Base 74, or Base 256) bit sequence is encoded (if the character map indicates at least one non-numeric character). This bit sequence represents the binary number resulting from a base-30 to Binary conversion (or a Base-74 to Binary conversion, or a direct transfer of Base-256 values) of the sequence of non-digit characters in the data (ignoring any intervening digits). Again, the number of encoded bits is directly determined from the number of non-numeric values being represented, as shown in Annex K. Note that if Base 256 was selected as the non-Numeric base, then the encoder is free to classify and encode each digit either as Base 10 or as Base 256 (Base 10 will be more efficient, unless outweighed by the ability to take advantage of a long Prefix or Suffix).

Note that an Alphanumeric subsection ends with several variable-length bit fields (the character map, and one or more Binary sections (representing the numeric and non-numeric Binary values). Note further that none of the lengths of these three variable-length bit fields are explicitly encoded (although one or two Extended-Base Binary segments may also be present, these have known lengths, determined from Prefix and/or Suffix runs). In order to determine the boundaries between these three variable-length fields, the decoder needs to implement a procedure, using knowledge of the remaining number of data bits, in order to correctly parse the Alphanumeric subsection. An example of such a procedure is described in Annex M.

I.8.2.5 Padding the last Byte

The last (least-significant) bit of the final Binary segment is also the last significant bit of the Packed Object. If there are any remaining bit positions in the last byte to be filled with pad bits, then the most significant pad bit shall be set to ‘1’, and any remaining less-significant pad bits shall be set to ‘0’. The decoder can determine the total number of non-pad bits in a Packed Object by examining the Length Section of the Packed Object (and if the Pad Indicator bit of that section is ‘1’, by also examining the last byte of the Packed Object).

I.9 ID Map and Directory encoding options

An ID Map can be more efficient than a list of ID Values, when encoding a relatively large number of ID Values. Additionally, an ID Map representation is advantageous for use in a Directory Packed Object. The ID Map itself (the first major subsection of every ID Map section) is structured identically whether in a Data or Directory IDMPO, but a Directory IDMPO’s ID Map section contains additional optional subsections. The structure of an ID Map section, containing one or more ID Maps, is described in section I.9.1, explained in terms of its usage in a Data IDMPO; subsequent sections explain the added structural elements in a Directory IDMPO.

I.9.1 ID Map Section structure

An IDMPO represents ID Values using a structure called an ID Map section, containing one or more ID Maps. Each ID Value encoded in a Data IDMPO is represented as a ‘1’ bit within an ID Map bit field, whose fixed length is equal to the number of entries in the corresponding Base Table. Conversely, each ‘0’ in the ID Map Field indicates the absence of the corresponding ID Value. Since the total number of ‘1’ bits within the ID Map Field equals the number of ID Values being represented, no explicit NumberOfIDs field is encoded. In order to implement the range of functionality made possible by this representation, the ID Map Section contains elements other than the ID Map itself. If present, the optional ID Map Section immediately follows the leading pattern indicating an IDMPO (as was described in I.4.2), and contains the following elements in the order listed below:

- An Application Indicator subsection (see I.5.3.1)
- an ID Map bit field (whose length is determined from the ID Size in the Application Indicator)
- a Full/Restricted Use bit (see I.5.3.2)
- (the above sequence forms an ID Map, which may optionally repeat multiple times)
- a Data/Directory indicator bit,
- an optional AuxMap section (never present in a Data IDMPO), and
- Closing Flag(s), consisting of an "Addendum Flag" bit. If '1', then an Addendum subsection is present at the end of the Object Info section (after the Object Length Information).

These elements, shown in Figure I 9-1 as a maximum structure (every element is present), are described in each of the next subsections.

Figure I 9-1: ID Map section

First ID Map		Optional additional ID Map(s)		Null App Indicator (single zero bit)	Data/ Directory Indicator Bit	(If directory) Optional AuxMap Section	Closing Flag Bit(s)
App Indicator	ID Map Bit Field (ends with F/R bit)	App Indicator	ID Map Field (ends with F/R bit)				
See I.5.3.1	See I.9.1.1 and I.5.3.2	As previous	As previous	See I.5.3.1		See Figure I 9-2	Addendum Flag Bit

When an ID Map section is encoded, it is always followed by an Object Length and Pad Indicator, and optionally followed by an Addendum subsection (all as have been previously defined), and then may be followed by any of the other sections defined for Packed Objects, except that a Directory IDMPO shall not include a Data section.

I.9.1.1 ID Map and ID Map bit field

An ID Map usually consists of an Application Indicator followed by an ID Map bit field, ending with a Full/Restricted Use bit. An ID Map bit field consists of a single "MapPresent" flag bit, then (if MapPresent is '1') a number of bits equal to the length determined from the ID Size pattern within the Application Indicator, plus one (the Full/Restricted Use bit). The ID Map bit field indicates the presence/absence of encoded data items corresponding to entries in a specific registered Primary or Alternate Base Table. The choice of base table is indicated by the encoded combination of DSFID and Application Indicator pattern that precedes the ID Map bit field. The MSB of the ID Map bit field corresponds to ID Value 0 in the base table, the next bit corresponds to ID Value 1, and so on.

In a Data Packed Object's ID Map bit field, each '1' bit indicates that this Packed Object contains an encoded occurrence of the data item corresponding to an entry in the registered Base Table associated with this ID Map. Note that the valid encoded entry may be found either in the first ("parentless") Packed Object of the chain (the one containing the ID Map) or in an Addendum IDLPO of that chain. Note further that one or more data entries may be encoded in an IDMPO, but marked "invalid" (by a Delete entry in an Addendum IDLPO).

An ID Map shall not correspond to a Secondary ID Table instead of a Base ID Table. Note that data items encoded in a "parentless" Data IDMPO shall appear in the same relative order in which they are listed in the associated Base Table. However, additional "out of order" data items may be added to an existing data IDMPO by appending an Addendum IDLPO to the Object.

An ID Map cannot indicate a specific number of instances (greater than one) of the same ID Value, and this would seemingly imply that only one data instance using a given ID Value can be encoded in a Data IDMPO. However, the ID Map method needs to support the case where more two or more encoded data items are from the same identifier "class" (and thus share the same ID Value). The following mechanisms address this need:

- Another data item of the same class can be encoded in an Addendum IDLPO of the IDMPO. Multiple occurrences of the same ID Value can appear on an ID List, each associated with different encoded values of the Secondary ID bits.
- A series of two or more encoded instances of the same "class" can be efficiently indicated by a single instance of an ID Value (or equivalently by a single ID Map bit), if the corresponding Base Table entry defines a "Repeat" Bit (see J.2.2).

An ID Map section may contain multiple ID Maps; a null Application Indicator section (with its AppIndicatorPresent bit set to '0') terminates the list of ID Maps.

I.9.1.2 Data/Directory and AuxMap indicator bits

A Data/Directory indicator bit is always encoded immediately following the last ID Map. By definition, a Data IDMPO has its Data/Directory bit set to '0', and a Directory IDMPO has its Data/Directory bit set to '1'. If the Data/Directory bit is set to '1', it is immediately followed by an AuxMap indicator bit which, if '1', indicates that an optional AuxMap section immediately follows.

I.9.1.3 Closing Flags bit(s)

The ID Map section ends with a single Closing Flag:

- The final bit of the Closing Flags is an Addendum Flag Bit which, if '1', indicates that there is an optional Addendum subsection encoded at the end of the Object Info section of the Packed Object. If present, the Addendum subsection is as described in Section I .5.6.

I.9.2 Directory Packed Objects

A "Directory Packed Object" is an IDMPO whose Directory bit is set to '1'. Its only inherent difference from a Data IDMPO is that it does not contain any encoded data items. However, additional mechanisms and usage considerations apply only to a Directory Packed Object, and these are described in the following subsections.

I.9.2.1 ID Maps in a Directory IDMPO

Although the structure of an ID Map is identical whether in a Data or Directory IDMPO, the semantics of the structure are somewhat different. In a Directory Packed Object's ID Map bit field, each '1' bit indicates that a Data Packed Object in the same data carrier memory bank contains a valid data item associated with the corresponding entry in the specified Base Table for this ID Map. Optionally, a Directory Packed Object may further indicate *which* Packed Object contains each data item (see the description of the optional AuxMap section below).

Note that, in contrast to a Data IDMPO, there is no required correlation between the order of bits in a Directory's ID Map and the order in which these data items are subsequently encoded in memory within a sequence of Data Packed Objects.

I.9.2.2 Optional AuxMap Section (Directory IDMPOs only)

An AuxMap Section optionally allows a Directory IDMPO's ID Map to indicate not only presence/absence of all the data items in this memory bank of the tag, but also which Packed Object encodes each data item. If the AuxMap indicator bit is '1', then an AuxMap section shall be encoded immediately after this bit. If encoded, the AuxMap section shall contain one PO Index Field for each of the ID Maps that precede this section. After the last PO Index Field, the AuxMap Section may optionally encode an ObjectOffsets list, where each ObjectOffset generally indicates the number of bytes from the start of the previous Packed Object to the start of the next Packed Object. This AuxMap structure is shown (for an example IDMPO with two ID Maps) in Figure I 9-2.

Figure I 9-2: Optional AuxMap section structure

<table>
<tr><td colspan="2">PO Index Field for first ID Map</td><td colspan="2">PO Index Field for second ID Map</td><td rowspan="2">Object Offsets Present bit</td><td colspan="5">Optional ObjectOffsets subsection</td></tr>
<tr><td>POindex Length</td><td>POindex Table</td><td>POindex Length</td><td>POindex Table</td><td>Object Offsets</td><td>Object1 offset</td><td>Object2 offset</td><td>…</td><td>ObjectN offset</td></tr>
</table>

					Multiplier	(EBV6)	(EBV6)		(EBV6)

Each PO Index Field has the following structure and semantics:

- A three-bit POindexLength field, indicating the number of index bits encoded for each entry in the PO Index Table that immediately follows this field (unless the POindex length is ‘000’, which means that no PO Index Table follows).
- A PO Index Table, consisting of an array of bits, one bit (or group of bits, depending on the POIndexLength) for every bit in the corresponding ID Map of this directory packed object. A PO Index Table entry (i.e., a “PO Index”) indicates (by relative order) which Packed Object contains the data item indicated by the corresponding ‘1’ bit in the ID Map. If an ID Map bit is '0', the corresponding PO Index Table entry is present but its contents are ignored.
- Every Packed Object is assigned an index value in sequence, without regard as to whether it is a “parentless” Packed Object or a “child” of another Packed Object, or whether it is a Data or Directory Packed Object.
- If the PO Index is within the first PO Index Table (for the associated ID Map) of the Directory “chain”, then:
    - a PO Index of zero refers to the first Packed Object in memory,
    - a value of one refers to the next Packed Object in memory, and so on
    - a value of *m*, where *m* is the largest value that can be encoded in the PO Index (given the number of bits per index that was set in the POindexLength), indicates a Packed Object whose relative index (position in memory) is *m or higher*. This definition allows Packed Objects higher than *m* to be indexed in an Addendum Directory Packed Object, as described immediately below. If no Addendum exists, then the precise position is either *m* or some indeterminate position greater than *m*.
- If the PO Index is not within the first PO Index Table of the directory chain for the associated ID Map (i.e., it is in an Addendum IDMPO), then:
    - a PO Index of zero indicates that a prior PO Index Table of the chain provided the index information,
    - a PO Index of *n* (*n* > 0) refers to the *nth* Packed Object above the highest index value available in the immediate parent directory PO; e.g., if the maximum index value in the immediate parent directory PO refers to PO number “3 or greater,” then a PO index of 1 in this addendum refers to PO number 4.
    - A PO Index of *m* (as defined above) similarly indicates a Packed Object whose position is the *mth* position, *or higher*, than the limit of the previous table in the chain.
- If the valid instance of an ID Value is in an Addendum Packed Object, an implementation may choose to set a PO Index to point directly to that Addendum, or may instead continue to point to the Packed Object in the chain that originally

contained the ID Value.

NOTE: The first approach sometimes leads to faster searching; the second sometimes leads to faster directory updates.

After the last PO Index Field, the AuxMap section ends with (at minimum) a single "ObjectOffsets Present" bit. A'0' value of this bit indicates that no ObjectOffsets subsection is encoded. If instead this bit is a '1', it is immediately followed by an ObjectOffsets subsection, which holds a list of EBV-6 "offsets" (the number of octets between the start of a Packed Object and the start of the next Packed Object). If present, the ObjectOffsets subsection consists of an ObjectOffsetsMultiplier followed by an Object Offsets list, defined as follows:

- An EBV-6 ObjectOffsetsMultiplier, whose value, when multiplied by 6, sets the total number of bits reserved for the entire ObjectOffsets list. The value of this multiplier should be selected to ideally result in sufficient storage to hold the offsets for the maximum number of Packed Objects that can be indexed by this Directory Packed Object's PO Index Table (given the value in the POIndexLength field, and given some estimated average size for those Packed Objects).

- a fixed-sized field containing a list of EBV-6 ObjectOffsets. The size of this field is exactly the number of bits as calculated from the ObjectOffsetsMultiplier. The first ObjectOffset represents the start of the second Packed Object in memory, relative to the first octet of memory (there would be little benefit in reserving extra space to store the offset of the *first* Packed Object). Each succeeding ObjectOffset indicates the start of the next Packed Object (relative to the previous ObjectOffset on the list), and the final ObjectOffset on the list points to the all-zero termination pattern where the *next* Packed Object may be written. An invalid offset of zero (EBV-6 pattern "000000") shall be used to terminate the ObjectOffset list. If the reserved storage space is fully occupied, it need not include this terminating pattern.

- In applications where the average Packed Object Length is difficult to predict, the reserved ObjectOffset storage space may sometimes prove to be insufficient. In this case, an Addendum Packed Object can be appended to the Directory Packed Object. This Addendum Directory Packed Object may contain null subsections for all but its ObjectOffsets subsection. Alternately, if it is anticipated that the capacity of the PO Index Table will also eventually be exceeded, then the Addendum Packed Object may also contain one or more non-null PO Index fields. Note that in a given instance of an AuxMap section, either a PO Index Table or an ObjectOffsets subsection may be the first to exceed its capacity. Therefore, the first position referenced by an ObjectOffsets list in an Addendum Packed Object need not coincide with the first position referenced by the PO Index Table of that same Addendum. Specifically, in an Addendum Packed Object, the first ObjectOffset listed is an offset referenced to the last ObjectOffset on the list of the "parent" Directory Packed Object.

I.9.2.3 Usage as a Presence/Absence Directory

In many applications, an Interrogator may choose to read the entire contents of any data carrier containing one or more "target" data items of interest. In such applications, the positional information of those data items within the memory is not needed during the initial reading operations; only a presence/absence indication is needed at this processing stage. An ID Map can form a particularly-efficient Presence/Absence directory for denoting the contents of a data carrier in such applications. A full directory structure encodes the offset or address (memory location) of every data element within the data carrier, which requires the writing of a large number of bits (typically 32 bits or more per data item). Inevitably, such an approach also requires reading a large number of bits over the air, just to determine whether an identifier of interest is present on a particular tag. In contrast, when only presence/absence information is needed, using an ID Map conveys the same information using only one bit per data item defined in the data system. The entire ID Map can be typically represented in 128 bits or less, and stays the same size as more data items are written to the tag.

A "Presence/Absence Directory" Packed Object is defined as a Directory IDMPO that does not contain a PO Index, and therefore provides no encoded information as to where individual data items reside within the data carrier. A Presence/Absence Directory can be converted to an "Indexed Directory" Packed Object (see I.9.2.4) by adding a PO Index in an Addendum Packed Object, as a "child" of the Presence/Absence Packed Object.

I.9.2.4 Usage as an Indexed Directory

In many applications involving large memories, an Interrogator may choose to read a Directory section covering the entire memory's contents, and then issue subsequent Reads to fetch the "target" data items of interest. In such applications, the positional information of those data items within the memory is important, but if many data items are added to a large memory over time, the directory itself can grow to an undesirable size.

An ID Map, used in conjunction with an AuxMap containing a PO Index, can form a particularly-efficient "Indexed Directory" for denoting the contents of an RFID tag, and their approximate locations as well. Unlike a full tag directory structure, which encodes the offset or address (memory location) of every data element within the data carrier, an Indexed Directory encodes a small relative position or index indicating which Packed Object contains each data element. An application designer may choose to also encode the locations of each Packed Object in an optional ObjectOffsets subsection as described above, so that a decoding system, upon reading the Indexed Directory alone, can calculate the start addresses of all Packed Objects in memory.

The utility of an ID Map used in this way is enhanced by the rule of most data systems that a given identifier may only appear once within a single data carrier. This rule, when an Indexed Directory is utilized with Packed Object encoding of the data in subsequent objects, can provide nearly-complete random access to reading data using relatively few directory bits. As an example, an ID Map directory (one bit per defined ID) can be associated with an additional AuxMap "PO Index" array (using, for example, three bits per defined ID). Using this arrangement, an interrogator would read the Directory Packed Object, and examine its ID Map to determine if the desired data item were present on the tag. If so, it would examine the 3 "PO Index" bits corresponding to that data item, to determine which of the first 8 Packed Objects on the tag contain the desired data item. If an optional ObjectOffsets subsection was encoded, then the Interrogator can calculate the starting address of the desired Packed Object directly; otherwise, the interrogator may perform successive read operations in order to fetch the desired Packed Object.

Appendix J Packed Objects ID Tables

J.1 Packed Objects Data Format registration file structure

A Packed Objects registered Data Format file consists of a series of “Keyword lines” and one or more ID Tables. Blank lines may occur anywhere within a Data Format File, and are ignored. Also, any line may end with extra blank columns, which are also ignored.

- A Keyword line consists of a Keyword (which always starts with “K-“) followed by an equals sign and a character string, which assigns a value to that Keyword. Zero or more space characters may be present on either side of the equals sign. Some Keyword lines shall appear only once, at the top of the registration file, and others may appear multiple times, once for each ID Table in the file.
- An ID Table lists a series of ID Values (as defined in I.5.3). Each row of an ID Table contains a single ID Value (in a required “IDvalue” column), and additional columns may associate Object IDs (OIDs), ID strings, Format strings, and other information with that ID Value. A registration file always includes a single “Primary” Base ID Table, zero or more “Alternate” Base ID Tables, and may also include one or more Secondary ID Tables (that are referenced by one or more Base ID Table entries).

To illustrate the file format, a hypothetical data system registration is shown in Figure J-1. In this hypothetical data system, each ID Value is associated with one or more OIDs and corresponding ID strings. The following subsections explain the syntax shown in the Figure.

Figure J- 1:Hypothetical Data Format registration file

K-Text = Hypothetical Data Format 100

K-Version = 1.0

K-TableID = F100B0

K-RootOID = urn:oid:1.0.12345.100

K-IDsize = 16

IDvalue	OIDs	IDstring	Explanation	FormatString
0	99	1Z	Legacy ID "1Z" corresponds to OID 99, is assigned IDval 0	14n
1	9%x30-33	7%x42-45	An OID in the range 90..93, Corresponding to ID 7B..7E	1*8an
2	(10)(20)(25)(37)	(A)(B)(C)(D)	a commonly-used set of IDs	(1n)(2n)(3n)(4n)
3	26/27	1A/2B	Either 1A or 2B is encoded, but not both	10n / 20n
4	(30) [31]	(2A) [3B]	2A is always encoded, optionally followed by 3B	(11n) [1*20n]
5	(40/41/42) (53) [55]	(4A/4B/4C) (5D) [5E]	One of A/B/C is encoded, then D, and optionally E	(1n/2n/3n) (4n) [5n]
6	(60/61/(64)[66])	(6A /6B / (6C) [6D])	Selections, one of which includes an Option	(1n / 2n / (3n][4n])

K-TableEnd = F100B0

J.1.1 File Header section

Keyword lines in the File Header (the first portion of every registration file) may occur in any order, and are as follows:

- **(Mandatory) K-Version = nn.nn**, which the registering body assigns, to ensure that any future revisions to their registration are clearly labeled.
- **(Optional) K-Interpretation = string**, where the "string" argument shall be one of the following: "ISO-646", "UTF-8", "ECI-nnnnnn" (where nnnnnn is a registered six-digit ECI number), ISO-8859-nn, or "UNSPECIFIED". The Default interpretation is "UNSPECIFIED". This keyword line allows non-default interpretations to be placed on the octets of data strings that are decoded from Packed Objects.
- **(Optional) K-ISO15434=nn**, where "nn" represents a Format Indicator (a two-digit numeric identifier) as defined in ISO/IEC 15434. This keyword line allows receiving

systems to optionally represent a decoded Packed Object as a fully-compliant ISO/IEC 15434 message. There is no default value for this keyword line.

- **(Optional) K-AppPunc = nn**, where nn represents (in decimal) the octet value of an ASCII character that is commonly used for punctuation in this application. If this keyword line is not present, the default Application Punctuation character is the hyphen.

In addition, comments may be included using the optional Keyword assignment line "K-text = string", and may appear zero or more times within a File Header or Table Header, but not in an ID Table body.

J.1.2 Table Header section

One or more Table Header sections (each introducing an ID Table) follow the File Header section. Each Table Header begins with a K-TableID keyword line, followed by a series of additional required and optional Keyword lines (which may occur in any order), as follows:

- **(Mandatory) K-TableID = FnnXnn**, where **Fnn** represents the ISO-assigned Data Format number (where 'nn' represents one or more decimal digits), and Xnn (where 'X' is either 'B' or 'S') is a registrant-assigned Table ID for each ID Table in the file. The first ID Table shall always be the Primary Base ID Table of the registration, with a Table ID of "B0". As many as seven additional "Alternate" Base ID Tables may be included, with higher sequential "Bnn" Table IDs. Secondary ID Tables may be included, with sequential Table IDs of the form "Snn".

- **(Mandatory) K-IDsize = nn**. For a base ID table, the value **nn** shall be one of the values from the "Maximum number of Table Entries" column of Table I 5-5. For a secondary ID table, the value **nn** shall be a power of two (even if not present in Table I 5-5.

- **(Optional) K-RootOID = urn:oid:i.j.k.ff** where:
  - **I, j, and k** are the leading arcs of the OID (as many arcs as required) and
  - **ff** is the last arc of the Root OID (typically, the registered Data Format number)

  If the K-RootOID keyword is not present, then the default Root OID is:
  - **urn:oid:1.0.15961.ff**, where "ff" is the registered Data Format number

- **Other optional Keyword lines:** in order to override the file-level defaults (to set different values for a particular table), a Table Header may invoke one or more of the Optional Keyword lines listed in for the File Header section.

The end of the Table Header section is the first non-blank line that does not begin with a Keyword. This first non-blank line shall list the titles for every column in the ID Table that immediately follows this line; column titles are case-sensitive.

An Alternate Base ID Table, if present, is identical in format to the Primary Base ID Table (but usually represents a smaller choice of identifiers, targeted for a specific application).

A Secondary ID Table can be invoked by a keyword in a Base Table's **OIDs** column. A Secondary ID Table is equivalent to a single Selection list (see J.3) for a single ID Value of a Base ID Table (except that a Secondary table uses K-Idsize to explicitly define the number of Secondary ID bits per ID); the IDvalue column of a Secondary table lists the value of the corresponding Secondary ID bits pattern for each row in the Secondary Table. An **OIDs** entry in a Secondary ID Table shall not itself contain a Selection list nor invoke another Secondary ID Table.

J.1.3 ID Table section

Each ID table consists of a series of one or more rows, each row including a mandatory "IDvalue" column, several defined Optional columns (such as "OIDs", "IDstring", and "FormatString"), and any number of Informative columns (such as the "Explanation" column in the hypothetical example shown above).

Each ID Table ends with a required Keyword line of the form:

- **K-TableEnd = FnnXnn**, where **FnnXnn** shall match the preceding **K-TableID** keyword line that introduced the table.

The syntax and requirements of all Mandatory and Optional columns shall be as described J.2.

J.2 Mandatory and Optional ID Table columns

Each ID Table in a Packed Objects registration shall include an IDvalue column, and may include other columns that are defined in this specification as Optional, and/or Informative columns (whose column heading is not defined in this specification).

J.2.1 IDvalue column (Mandatory)

Each ID Table in a Packed Objects registration shall include an IDvalue column. The ID Values on successive rows shall increase monotonically. However, the table may terminate before reaching the full number of rows indicated by the Keyword line containing **K-IDsize**. In this case, a receiving system will assume that all remaining ID Values are reserved for future assignment (as if the OIDs column contained the keyword "K-RFA"). If a registered Base ID Table does not include the optional OIDs column described below, then the IDvalue shall be used as the last arc of the OID.

J.2.2 OIDs and IDstring columns (Optional)

A Packed Objects registration always assigns a final OID arc to each identifier (either a number assigned in the "OIDs" column as will be described below, or if that column is absent, the IDvalue is assigned as the default final arc). The OIDs column is required rather than optional, if a single IDvalue is intended to represent either a combination of OIDs or a choice between OIDs (one or more Secondary ID bits are invoked by any entry that presents a choice of OIDs).

A Packed Objects registration may include an IDString column, which if present assigns an ASCII-string name for each OID. If no name is provided, systems must refer to the identifier by its OID (see J.4). However, many registrations will be based on data systems that do have an ASCII representation for each defined Identifier, and receiving systems may optionally output a representation based on those strings. If so, the ID Table may contain a column indicating the IDstring that corresponds to each OID. An empty IDstring cell means that there is no corresponding ASCII string associated with the OID. A non-empty IDstring shall provide a name for every OID invoked by the OIDs column of that row (or a single name, if no OIDs column is present). Therefore, the sequence of combination and selection operations in an IDstring shall exactly match those in the row's OIDs column.

A non-empty **OIDs** cell may contain either a keyword, an ASCII string representing (in decimal) a single OID value, or a compound string (in ABNF notation) that a defines a choice and/or a combination of OIDs. The detailed syntax for compound OID strings in this column (which also applies to the IDstring column) is as defined in section J.3. Instead of containing a simple or compound OID representation, an OIDs entry may contain one of the following Keywords:

- **K-Verbatim = OIDddBnn**, where "dd" represents the chosen penultimate arc of the OID, and "Bnn" indicates one of the Base 10, Base 40, or Base 74 encoding tables. This entry invokes a number of Secondary ID bits that serve two purposes:
    - They encode an ASCII identifier "name" that might not have existed at the time the table was registered. The name is encoded in the Secondary ID bits section as a series of Base-n values representing the ASCII characters of the name, preceded by a four-bit field indicating the number of Base-n values that follow (zero is permissible, in order to support RFA entries as described below).
    - The cumulative value of these Secondary ID bits, considered as a single unsigned binary integer and converted to decimal, is the final "arc" of the OID for this "verbatim-encoded' identifier.
- **K-Secondary = Snn**, where "Snn" represents the Table ID of a Secondary ID Table in the same registration file. This is equivalent to a Base ID Table row OID entry that contains a single Selection list (with no other components at the top level), but instead of listing these components in the Base ID Table, each component is listed as a separate row in the Secondary ID Table, where each may be assigned a unique OID, ID string, and FormatString.
- **K-Proprietary=OIDddPnn**, where nn represents a fixed number of Secondary ID bits that encode an optional Enterprise Identifier indicating who wrote the proprietary data (an entry of **K-Proprietary=OIDddP0** indicates an "anonymous" proprietary data item).
- **K-RFA = OIDddBnn**, where "Bnn" is as defined above for Verbatim encoding, except that "B0" is a valid assignment (meaning that no Secondary ID bits are invoked). This keyword represents a Reserved for Future Assignment entry, with an option for Verbatim encoding of the Identifier "name" once a name is assigned by the entity who registered this Data Format. Encoders may use this entry, with a four-bit "verbatim" length of zero, until an Identifier "name" is assigned. A specific

FormatString may be assigned to K-RFA entries, or the default a/n encoding may be utilized.

Finally, any OIDs entry may end with a single "**R**" character (preceded by one or more space characters), to indicate that a "Repeat" bit shall be encoded as the last Secondary ID bit invoked by the entry. If '1', this bit indicates that another instance of this class of identifier is also encoded (that is, this bit acts as if a repeat of the ID Value were encoded on an ID list). If '1', then this bit is followed by another series of Secondary ID bits, to represent the particulars of this additional instance of the ID Value.

An IDstring column shall not contain any of the above-listed Keyword entries, and an IDstring entry shall be empty when the corresponding OIDs entry contains a Keyword.

J.2.3 FormatString column (Optional)

An ID Table may optionally define the data characteristics of the data associated with a particular identifier, in order to facilitate data compaction. If present, the FormatString entry specifies whether a data item is all-numeric or alphanumeric (i.e., may contain characters other than the decimal digits), and specifies either a fixed length or a variable length. If no FormatString entry is present, then the default data characteristic is alphanumeric. If no FormatString entry is present, or if the entry does not specify a length, then any length >=1 is permitted. Unless a single fixed length is specified, the length of each encoded data item is encoded in the Aux Format section of the Packed Object, as specified in I.7.

If a given IDstring entry defines more than a single identifier, then the corresponding FormatString column shall show a format string for each such identifier, using the same sequence of punctuation characters (disregarding concatenation) as was used in the corresponding IDstring.

The format string for a single identifier shall be one of the following:

- A length qualifier followed by "n" (for always-numeric data);
- A length qualifier followed by "an" (for data that may contain non-digits); or
- A fixed-length qualifier, followed by "n", followed by one or more space characters, followed by a variable-length qualifier, followed by "an".

A length qualifier shall be either null (that is, no qualifier present, indicating that any length >= 1 is legal), a single decimal number (indicating a fixed length) or a length range of the form "i*j", where "I" represents the minimum allowed length of the data item, "j" represents the maximum allowed length, and i <= j. In the latter case, if "j" is omitted, it means the maximum length is unlimited.

Data corresponding to an "n" in the FormatString are encoded in the KLN subsection; data corresponding to an "an" in the FormatString are encoded in the A/N subsection.

When a given instance of the data item is encoded in a Packed Object, its length is encoded in the Aux Format section as specified in I.7.2. The minimum value of the range is not itself encoded, but is specified in the ID Table's FormatString column.

Example:

A FormatString entry of “3*6n” indicates an all-numeric data item whose length is always between three and six digits inclusive. A given length is encoded in two bits, where ‘00’ would indicate a string of digits whose length is “3”, and ‘11’ would indicate a string length of six digits.

J.2.4 Interp column (Optional)

Some registrations may wish to specify information needed for output representations of the Packed Object’s contents, other than the default OID representation of the arcs of each encoded identifier. If this information is invariant for a particular table, the registration file may include keyword lines as previously defined. If the interpretation varies from row to row within a table, then an Interp column may be added to the ID Table. This column entry, if present, may contain one or more of the following keyword assignments (separated by semicolons), as were previously defined (see J.1.1 and J.1.2):

- **K-RootOID = urn:oid:i.j.k.l…**
- **K-Interpretation = string**
- **K-ISO15434=nn**

If used, these override (for a particular Identifier) the default file-level values and/or those specified in the Table Header section.

J.3 Syntax of OIDs, IDstring, and FormatString Columns

In a given ID Table entry, the OIDs, IDString, and FormatString column may indicate one or more mechanisms described in this section. J.3.1 specifies the formal grammar for these columns, and the meaning is described below. In the descriptions below, the word “Identifier” means either an OID final arc (in the context of the OIDs column) or an IDString name (in the context of the IDstring column). If both columns are present, only the OIDs column actually invokes Secondary ID bits.

- A ***Single component*** resolving to a single Identifier, in which case no additional Secondary ID bits are invoked.
- (For OIDs and IDString columns only) A single component resolving to one of a series of closely-related Identifiers, where the Identifier’s string representation varies only at one or more character positions. This is indicated using the ***Concatenation*** operator ‘%’ to introduce a range of ASCII characters at a specified position. For example, an OID whose final arc is defined as “391n”, where the fourth digit ‘n’ can be any digit from ‘0’ to ‘6’ (ASCII characters 30_{hex} to 36_{hex} inclusive) is represented by the component **391%x30-36** (note that no spaces are allowed) A Concatenation invokes the minimum number of Secondary ID digits needed to indicate the specified range. When both an OIDs column and an IDstring column are populated for a given row, both shall contain the same number of concatations, with the same ranges (so that the numbers and values of Secondary ID bits invoked are consistent). However, the minimum value listed for the two ranges can differ, so that (for example) the OID’s digit can range from 0 to 3, while the corresponding IDstring character can range from “B” to “E” if so desired. Note that the use of Concatenation inherently

constrains the relationship between OID and IDString, and so Concatenation may not be useable under all circumstances (the Selection operation described below usually provides an alternative).

- A ***Combination*** of two or more identifier components in an ordered sequence, indicated by surrounding each component of the sequence with parentheses. For example, an IDstring entry **(A)(%x30-37B)(2C)** indicates that the associated ID Value represents a sequence of the following three identifiers:

  - Identifier “A”, then

  - An identifier within the range “0B” to “7B” (invoking three Secondary ID bits to represent the choice of leading character), then

  - Identifier “2C

  Note that a Combination does not itself invoke any Secondary ID bits (unless one or more of its components do).

- An ***Optional*** component is indicated by surrounding the component in brackets, which may viewed as a “conditional combination.” For example the entry (A)[B][C][D] indicates that the ID Value represents identifier A, optionally followed by B, C, and/or D. A list of Options invokes one Secondary ID bit for each component in brackets, wherein a ‘1’ indicates that the optional component was encoded.

- A ***Selection*** between several mutually-exclusive components is indicated by separating the components by forward slash characters. For example, the IDstring entry **(A/B/C/(D)(E))** indicates that the fully-qualified ID Value represents a single choice from a list of four choices (the fourth of which is a Combination). A Selection invokes the minimum number of Secondary ID bits needed to indicate a choice from a list of the specified number of components.

In general, a “compound” OIDs or IDstring entry may contain any or all of the above operations. However, to ensure that a single left-to-right parsing of an OIDs entry results in a deterministic set of Secondary ID bits (which are encoded in the same left-to-right order in which they are invoked by the OIDs entry), the following restrictions are applied:

- A given Identifier may only appear once in an OIDs entry. For example, the entry (A)(B/A) is invalid

- A OIDs entry may contain at most a single Selection list

- There is no restriction on the number of Combinations (because they invoke no Secondary ID bits)

- There is no restriction on the total number of Concatenations in an OIDs entry, but no single Component may contain more than two Concatenation operators.

- An Optional component may be a component of a Selection list, but an Optional component may not be a compound component, and therefore shall not include a Selection list nor a Combination nor Concatenation.

- A OIDs or IDstring entry may not include the characters '(', ')', '[', ']', '%', '-', or '/', unless used as an Operator as described above. If one of these characters is part of a defined data system Identifier "name", then it shall be represented as a single literal Concatenated character.

J.3.1 Formal Grammar for OIDs, IDString, and FormatString Columns

In each ID Table entry, the contents of the OIDs, IDString, and FormatString columns shall conform to the following grammar for `Expr`, unless the column is empty or (in the case of the OIDs column) it contains a keyword as specified in J.2.2. All three columns share the same grammar, except that the syntax for `COMPONENT` is different for each column as specified below. In a given ID Table Entry, the contents of the OIDs, IDString, and FormatString column (except if empty) shall have identical parse trees according to this grammar, except that the `COMPONENT`s may be different. Space characters are permitted (and ignored) anywhere in an `Expr`, except that in the interior of a `COMPONENT` spaces are only permitted where explicitly specified below.

```
Expr ::= SelectionExpr | "(" SelectionExpr ")" | SelectionSubexpr

SelectionExpr ::= SelectionSubexpr ( "/" SelectionSubexpr )+

SelectionSubexpr ::= COMPONENT | ComboExpr

ComboExpr ::= ComboSubexpr+

ComboSubexpr ::= "(" COMPONENT ")" | "[" COMPONENT "]"
```

For the OIDs column, `COMPONENT` shall conform to the following grammar:

```
COMPONENT_OIDs ::= (COMPONENT_OIDs_Char | Concat)+

COMPONENT_OIDs_Char ::= ("0".."9")+
```

For the IDString column, `COMPONENT` shall conform to the following grammar:

```
COMPONENT_IDString ::= UnquotedIDString | QuotedIDString

UnquotedIDString ::= (UnQuotedIDStringChar | Concat)+

UnquotedIDStringChar ::=
   "0".."9" | "A".."Z" | "a".."z" | "_"

QuotedIDString ::= QUOTE QuotedIDStringConstituent+ QUOTE

QuotedIDStringConstituent ::=
    " " | "!" | "#".."~" | (QUOTE QUOTE)
```

`QUOTE` refers to ASCII character 34 (decimal), the double quote character.

When the `QuotedIDString` form for `COMPONENT_IDString` is used, the beginning and ending `QUOTE` characters shall *not* be considered part of the IDString.

Between the beginning and ending `QUOTE`, all ASCII characters in the range 32 (decimal) through 126 (decimal), inclusive, are allowed, except that two `QUOTE` characters in a row shall denote a single double-quote character to be included in the IDString.

In the `QuotedIDString` form, a `%` character does not denote the concatenation operator, but instead is just a percent character included literally in the IDString. To use the concatenation operator, the `UnquotedIDString` form must be used. In that case, a degenerate concatenation operator (where the start character equals the end character) may be used to include a character into the IDString that is not one of the characters listed for `UnquotedIDStringChar`.

For the FormatString column, `COMPONENT` shall conform to the following grammar:

```
COMPONENT_FormatString ::= Range? (“an” | “n”)
                         | FixedRange “n” “ ”+ VarRange “an”

Range ::= FixedRange | VarRange

FixedRange ::= Number

VarRange ::= Number “*” Number?

Number ::= (“0”..“9”)+
```

The syntax for `COMPONENT` for the OIDs and IDString columns make reference to `Concat`, whose syntax is specified as follows:

```
Concat ::= “%” “x” HexChar HexChar “-” HexChar HexChar

HexChar ::= (“0”..“9” | “A”..“F”)
```

The hex value following the hyphen shall be greater than or equal to the hex value preceding the hyphen. In the OIDs column, each hex value shall be in the range 30_{hex} to 39_{hex}, inclusive. In the IDString column, each hex value shall be in the range 20_{hex} to $7E_{hex}$, inclusive.

J.4 OID input/output representation

The default method for representing the contents of a Packed Object to a receiving system is as a series of name/value pairs, where the name is an OID, and the value is the decoded data string associated with that OID. Unless otherwise specified by a **K-RootOID** keyword line, the default root OID is **urn:oid:1.0.15961.ff,** where **ff** is the Data Format encoded in the DSFID. The final arc of the OID is (by default) the IDvalue, but this is typically overridden by an entry in the OIDs column. Note that an encoded Application Indicator (see I.5.3.1) may change **ff** from the value indicated by the DSFID.

If supported by information in the ID Table’s IDstring column, a receiving system may translate the OID output into various alternative formats, based on the IDString representation of the OIDs. One such format, as described in ISO/IEC 15434, requires as additional information a two-digit Format identifier; a table registration may provide this information using the **K-ISO15434** keyword as described above.

The combination of the K-RootOID keyword and the OIDs column provides the registering entity an ability to assign OIDs to data system identifiers without regard to how they are actually encoded, and therefore the same OID assignment can apply regardless of the access method.

J.4.1 “ID Value OID” output representation

If the receiving system does not have access to the relevant ID Table (possibly because it is newly-registered), the Packed Objects decoder will not have sufficient information to convert the IDvalue (plus Secondary ID bits) to the intended OID. In order to ease the introduction of new or external tables, encoders have an option to follow “restricted use” rules (see I.5.3.2).

When a receiving system has decoded a Packed Object encoded following “restricted use” rules, but does not have access to the indicated ID Table, it shall construct an “ID Value OID” in the following format:

urn:oid:1.0.15961.300.ff.bb.idval.secbits

where **1.0.15961.300** is a Root OID with a reserved Data Format of “300” that is never encoded in a DSFID, but is used to distinguish an “ID Value OID” from a true OID (as would have been used if the ID Table were available). The reserved value of 300 is followed by the encoded table’s Data Format (**ff**) (which may be different from the DSFID’s default), the table ID (**bb**) (always ‘0’, unless otherwise indicated via an encoded Application Indicator), the encoded ID value, and the decimal representation of the invoked Secondary ID bits. This process creates a unique OID for each unique fully-qualified ID Value. For example, using the hypothetical ID Table shown in Annex L (but assuming, for illustration purposes, that the table’s specified Root OID is **urn:oid:1.0.12345.9**, then an “AMOUNT” ID with a fourth digit of ‘2’ has a true OID of:

urn:oid:1.0.12345.9.3912

and an “ID Value OID” of

urn:oid:1.0.15961.300.9.0.51.2

When a single ID Value represents multiple component identifiers via combinations or optional components, their multiple OIDs and data strings shall be represented separately, each using the same “ID Value OID” (up through and including the Secondary ID bits arc), but adding as a final arc the component number (starting with “1” for the first component decoded under that IDvalue).

If the decoding system encounters a Packed Object that references an ID Table that is unavailable to the decoder, but the encoder chose not to set the “Restricted Use” bit in the Application Indicator, then the decoder shall either discard the Packed Object, or relay the entire Packed Object to the receiving system as a single undecoded binary entity, a sequence of octets of the length specified in the ObjectLength field of the Packed Object. The OID for an undecoded Packed Object shall be **urn:oid:1.0.15961.301.ff.n**, where

"301" is a Data Format reserved to indicate an undecoded Packed Object, "ff" shall be the Data Format encoded in the DSFID at the start of memory, and an optional final arc 'n' may be incremented sequentially to distinguish between multiple undecoded Packed Objects in the same data carrier memory.

Appendix K Packed Objects Encoding tables

Packed Objects primarily utilize two encoding bases:

- Base 10, which encodes each of the digits '0' through '9' in one Base 10 value
- Base 30, which encodes the capital letters and selectable punctuation in one Base-30 value, and encodes punctuation and control characters from the remainder of the ASCII character set in two base-30 values (using a Shift mechanism)

For situations where a high percentage of the input data's non-numeric characters would require pairs of base-30 values, two alternative bases, Base 74 and Base 256, are also defined:

- The values in the Base 74 set correspond to the invariant subset of ISO 646 (which includes the GS1 character set), but with the digits eliminated, and with the addition of GS and <space> (GS is supported for uses other than as a data delimiter).
- The values in the Base 256 set may convey octets with no graphical-character interpretation, or "extended ASCII values" as defined in ISO 8859-6, or UTF-8 (the interpretation may be set in the registered ID Table for an application). The characters '0' through '9' (ASCII values 48 through 57) are supported, and an encoder may therefore encode the digits either by using a prefix or suffix (in Base 256) or by using a character map (in Base 10). Note that in GS1 data, FNC1 is represented by ASCII <GS> (octet value 29_{dec}).

Finally, there are situations where compaction efficiency can be enhanced by run-length encoding of base indicators, rather than by character map bits, when a long run of characters can be classified into a single base. To facilitate that classification, additional "extension" bases are added, only for use in Prefix and Suffix Runs.

- In order to support run-length encoding of a primarily-numeric string with a few interspersed letters, a Base 13 is defined, per Table B-2
- Two of these extension bases (Base 40 and Base 84) are simply defined, in that they extend the corresponding non-numeric bases (Base 30 and Base 74, respectively) to also include the ten decimal digits. The additional entries, for characters '0' through '9', are added as the next ten sequential values (values 30 through 39 for Base 40, and values 74 through 83 for Base 84).
- The "extended" version of Base 256 is defined as Base 40. This allows an encoder the option of encoding a few ASCII control or upper-ASCII characters in Base 256, while using a Prefix and/or Suffix to more efficiently encode the remaining non-numeric characters.

The number of bits required to encode various numbers of Base 10, Base 16, Base 30, Base 40, Base 74, and Base 84 characters are shown in Figure B-1. In all cases, a limit is placed on the size of a single input group, selected so as to output a group no larger than 20 octets.

Figure K-1: Required number of bits for a given number of Base ‘N’ values

```
/* Base10 encoding accepts up to 48 input values per group: */
static const unsigned char bitsForNumBase10[] = {
/*  0 -  9 */    0,   4,   7,  10,  14,  17,  20,  24,  27,  30,
/* 10 - 19 */   34,  37,  40,  44,  47,  50,  54,  57,  60,  64,
/* 20 - 29 */   67,  70,  74,  77,  80,  84,  87,  90,  94,  97,
/* 30 - 39 */  100, 103, 107, 110, 113, 117, 120, 123, 127, 130,
/* 40 - 48 */  133, 137, 140, 143, 147, 150, 153, 157, 160};

/* Base13 encoding accepts up to 43 input values per group: */
static const unsigned char bitsForNumBase13[] = {
/*  0 -  9 */    0,   4,   8,  12,  15,  19,  23,  26,  30,  34,
/* 10 - 19 */   38,  41,  45,  49,  52,  56,  60,  63,  67,  71,
/* 20 - 29 */   75,  78,  82,  86,  89,  93,  97, 100, 104, 108,
/* 30 - 39 */  112, 115, 119, 123, 126, 130, 134, 137, 141, 145,
/* 40 - 43 */  149, 152, 156, 160 };

/* Base30 encoding accepts up to 32 input values per group: */
static const unsigned char bitsForNumBase30[] = {
/*  0 -  9 */    0,   5,  10,  15,  20,  25,  30,  35,  40,  45,
/* 10 - 19 */   50,  54,  59,  64,  69,  74,  79,  84,  89,  94,
/* 20 - 29 */   99, 104, 108, 113, 118, 123, 128, 133, 138, 143,
/* 30 - 32 */  148, 153, 158};

/* Base40 encoding accepts up to 30 input values per group: */
static const unsigned char bitsForNumBase40[] = {
/*  0 -  9 */    0,   6,  11,  16,  22,  27,  32,  38,  43,  48,
/* 10 - 19 */   54,  59,  64,  70,  75,  80,  86,  91,  96, 102,
/* 20 - 29 */  107, 112, 118, 123, 128, 134, 139, 144, 150, 155,
/* 30  */  160 };

/* Base74 encoding accepts up to 25 input values per group: */
static const unsigned char bitsForNumBase74[] = {
/*  0 -  9 */    0,   7,  13,  19,  25,  32,  38,  44,  50,  56,
/* 10 - 19 */   63,  69,  75,  81,  87,  94, 100, 106, 112, 118,
/* 20 - 25 */  125, 131, 137, 143, 150, 156 };

/* Base84 encoding accepts up to 25 input values per group: */
static const unsigned char bitsForNumBase84[] = {
/*  0 -  9 */    0,   7,  13,  20,  26,  32,  39,  45,  52,  58,
/* 10 - 19 */   64,  71,  77,  84,  90,  96, 103, 109, 116, 122,
/* 20 - 25 */  128, 135, 141, 148, 154, 160 };
```

Table K-1: Base 30 Character set

Val	Basic set		Shift 1 set		Shift 2 set	
	Char	Decimal	Char	Decimal	Char	Decimal
0	A-Punc[1]	N/A	NUL	0	space	32
1	A	65	SOH	1	!	33
2	B	66	STX	2	“	34
3	C	67	ETX	3	#	35
4	D	68	EOT	4	$	36
5	E	69	ENQ	5	%	37
6	F	70	ACK	6	&	38
7	G	71	BEL	7	‘	39
8	H	72	BS	8	(	40
9	I	73	HT	9	)	41
10	J	74	LF	10	*	42
11	K	75	VT	11	+	43
12	L	76	FF	12	,	44
13	M	77	CR	13	-	45
14	N	78	SO	14	.	46
15	O	79	SI	15	/	47
16	P	80	DLE	16	:	58
17	Q	81	ETB	23	;	59
18	R	82	ESC	27	<	60
19	S	83	FS	28	=	61
20	T	84	GS	29	>	62
21	U	85	RS	30	?	63
22	V	86	US	31	@	64
23	W	87	invalid	N/A	\	92
24	X	88	invalid	N/A	^	94
25	Y	89	invalid	N/A	_	95
26	Z	90	[	91	‘	96
27	Shift 1	N/A	]	93	|	124
28	Shift 2	N/A	{	123	~	126
29	P-Punc[2]	N/A	}	125	invalid	N/A

Note 1: **Application-Specified Punctuation** character (Value 0 of the Basic set) is defined by default as the ASCII hyphen character (45_{dec}), but may be redefined by a registered Data Format

Note 2: **Programmable Punctuation** character (Value 29 of the Basic set): the first appearance of P-Punc in the alphanumeric data for a packed object, whether that first appearance is compacted into the Base 30 segment or the Base 40 segment, acts as a <Shift 2>, and also “programs” the character to be represented by second and subsequent appearances of P-Punc (in either segment) for the remainder of the alphanumeric data in that packed object. The Base 30 or Base 40 value immediately following that first appearance is interpreted using the Shift 2 column (Punctuation), and assigned to subsequent instances of P-Punc for the packed object.

Table K-2: Base 13 Character set

Value	Basic set		Shift 1 set		Shift 2 set		Shift 3 set	
	Char	Decimal	Char	Decimal	Char	Decimal	Char	Decimal
0	0	48	A	65	N	78	space	32
1	1	49	B	66	O	79	$	36
2	2	50	C	67	P	80	%	37
3	3	51	D	68	Q	81	&	38
4	4	52	E	69	R	82	*	42
5	5	53	F	70	S	83	+	43
6	6	54	G	71	T	84	,	44
7	7	55	H	72	U	85	-	45
8	8	56	I	73	V	86	.	46
9	9	57	J	74	W	87	/	47
10	Shift1	N/A	K	75	X	88	?	63
11	Shift2	N/A	L	76	Y	89	_	95
12	Shift3	N/A	M	77	Z	90	<GS>	29

Table K-3: Base 40 Character set

Val	Basic set		Shift 1 set		Shift 2 set	
	Char	Decimal	Char	Decimal	Char	Decimal
0	See Table K-1					
…	…					
29	See Table K-1					
30	0	48				
31	1	49				
32	2	50				
33	3	51				
34	4	52				
35	5	53				
36	6	54				
37	7	55				
38	8	56				
39	9	57				

Table K-4: Base 74 Character Set

Val	Char	Decimal	Val	Char	Decimal	Val	Char	Decimal
0	GS	29	**25**	F	70	**50**	d	100
1	!	33	**26**	G	71	**51**	e	101
2	"	34	**27**	H	72	**52**	f	102
3	%	37	**28**	I	73	**53**	g	103

4	&	38	**29**	J	74	**54**	h	104
5	'	39	**30**	K	75	**55**	i	105
6	(	40	**31**	L	76	**56**	j	106
7	)	41	**32**	M	77	**57**	k	107
8	*	42	**33**	N	78	**58**	l	108
9	+	43	**34**	O	79	**59**	m	109
10	,	44	**35**	P	80	**60**	n	110
11	-	45	**36**	Q	81	**61**	o	111
12	.	46	**37**	R	82	**62**	p	112
13	/	47	**38**	S	83	**63**	q	113
14	:	58	**39**	T	84	**64**	r	114
15	;	59	**40**	U	85	**65**	s	115
16	<	60	**41**	V	86	**66**	t	116
17	=	61	**42**	W	87	**67**	u	117
18	>	62	**43**	X	88	**68**	v	118
19	?	63	**44**	Y	89	**69**	w	119
20	A	65	**45**	Z	90	**70**	x	120
21	B	66	**46**	_	95	**71**	y	121
22	C	67	**47**	a	97	**72**	z	122
23	D	68	**48**	b	98	**73**	Space	32
24	E	69	**49**	c	99			

Table K-5: Base 84 Character Set

Val	Char	Decimal	Val	Char	Decimal	Val	Char	Decimal
0	FNC1	N/A	**25**	F		**50**	d	
1-73	See Table K-4							
74	0	48	**78**	4	52	**82**	8	56
75	1	49	**79**	5	53	**83**	9	57
76	2	50	**80**	6	54			
77	3	51	**81**	7	55			

Appendix L Encoding Packed Objects (non-normative)

In order to illustrate a number of the techniques that can be invoked when encoding a Packed Object, the following sample input data consists of data elements from the GS1 Application Identifier (AI) data system. This data represents:

- An Expiration date (AI 17) of October 31, 2006
- An Amount Payable (AI 391n) of 1234.56 Euros ("978" is the ISO Country Code which will indicate that the amount payable is in Euros)
- A Lot Number (AI 10) of 1A23B456CD, and

The application will present the above input to the encoder as a list of OID/Value pairs. The resulting input data, represented below as a single data string (wherein each AI is shown in parentheses) is:

(17)061031(3912)978123456(10)1A23B456CD

The example will use a hypothetical ID Table based on GS1 Application Identifiers. In this hypothetical table, each ID Value is a seven-bit index into the Base ID Table; the entries relevant to this example are shown in Table L-1.

Encoding is performed in the following steps:

- Three AI's are to be encoded, using Table L-1.
- As shown in the table's IDstring column, the combination of AI 17 and AI 10 is efficiently supported (because it is commonly seen in applications), and thus the encoder re-orders the input so that 17 and 10 are adjacent and in the order indicated in the IDString column:
    - (17)061031(10)1A23B456CD(3912)978123456

  Now, this AI pair can be assigned a single ID Value of 125 (decimal). The FormatString column for this entry shows that the encoded data will always consist of a fixed-length 6-digit string, followed by a variable-length alphanumeric string.

- Also as shown in Table L-1, AI 391n has an ID Value of 51(decimal). The IDstring entry for this AI shows that the AI string is formed by concatenating "391" with a suffix consisting of a single character in the range 30_{hex} to 39_{hex} (i.e., a decimal digit). Since that is a range of ten possibilities, a four-bit number will need to be encoded in the Secondary ID section to indicate which suffix character was chosen. The FormatString column for this entry shows that its data is variable-length numeric; the variable length information will require four bits to be encoded in the Aux Format section.
- Since only a small percentage of the 128-entry ID Table is utilized in this Packed Object, the encoder chooses an ID List format, rather than an ID Map format. As this is the default format, no Format Flags section is required.
- This results in the following Object Info section:
  - EBV-6 (ObjectLength): the value is TBD at this stage of the encoding process
  - Pad Indicator bit: TBD at this stage
  - EBV-3 (numberOfIDs) of 001 (meaning two ID Values will follow)
  - An ID List, including:
    - First ID Value: 125 (dec) in 7 bits, representing AI 17 followed by AI 10
    - Second ID Value: 51(decimal) in 7 bits, representing AI 391n
- A Secondary ID section is encoded as '0010', indicating the trailing '2' of the 391n AI. In the GS1 definition of this AI, a fourth AI digit of '2' means that two digits follow the implied decimal point, but that information is not needed in order to encode or decode the Packed Object.
- Next, an Aux Format section is encoded. An initial '1' bit is encoded, invoking the Packed-Object compaction method. Of the three AIs, only AI (391n) requires encoded Aux Format information: a four-bit pattern of '0101' (representing "six" variable-length digits – as "one" is the first allowed choice, a pattern of "0101" denotes "six").
- Next, the encoder encodes the first data item, for AI 17, which is defined as a fixed-length six-digit data item. The six digits of the source data string are "061031", which are converted to a sequence of six Base-10 values by subtracting 30_{hex} from each character of the string (the resulting values are denoted as values v_5 through v_0 in the formula below). These are then converted to a single Binary value, using the following formula:
  - $10^5 * v_5 + 10^4 * v_4 + 10^3 * v_3 + 10^2 * v_2 + 10^1 * v_1 + 10^0 * v_0$

  According to Figure K-1, a six-digit number is always encoded into 20 bits (regardless of any leading zero's in the input), resulting in a Binary string of:

  "0000 11101110 01100111"

- The next data item is for AI 10, but since the table indicates that this AI's data is alphanumeric, encoding into the Packed Object is deferred until after all of the known-length numeric data is encoded.
- Next, the encoder finds that one of the three AI's, AI 391n, is defined by Table D-1 as all-numeric, whose length of 9 (in this example) was encoded as (9 – 4 = 5) into four bits within the Aux Format subsection. Thus, a Known-Length-Numeric subsection is encoded for this data item, consisting of a binary value bit-pattern encoding 9 digits. Using Figure K-1 in Annex K, the encoder determines that 30 bits need to be encoded in order to represent a 9-digit number as a binary value. In this example, the binary value equivalent of "978123456" is the 30-bit binary sequence:

  "111010010011001111101011000000"
- At this point, encoding of the Known-Length Numeric subsection of the Data Section is complete.

Note that, so far, the total number of encoded bits is (3 + 6 + 1 + 7 + 7 + 4 + 5 + 20 + 30) or 83 bits, representing the IDLPO Length Section (assuming that a single EBV-6 vector remains sufficient to encode the Packed Object's length), two 7-bit ID Values, the Secondary ID and Aux Format sections, and two Known-Length-Numeric compacted binary fields.

At this stage, only one non-numeric AI data string (for AI 10) remains to be encoded in the Alphanumeric subsection. The 10-character source data string is "1A23B456CD". This string contains no characters requiring a base-30 Shift out of the basic Base-30 character set, and so Base-30 is selected for the non-numeric base (and so the first bit of the Alphanumeric subsection is set to '0' accordingly). The data string has no substrings with six or more successive characters from the same base, and so the next two bits are set to '00' (indicating that neither a Prefix nor a Suffix is run-length encoded). Thus, a full 10-bit Character Map needs to be encoded next. Its specific bit pattern is '0100100011', indicating the specific sequence of digits and non-digits in the source data string "1A23B456CD".

Up to this point, the Alphanumeric subsection contains the 13-bit sequence '0 00 0100100011'. From Annex K, it can be determined that lengths of the two final bit sequences (encoding the Base-10 and Base-30 components of the source data string) are 20 bits (for the six digits) and 20 bits (for the four uppercase letters using Base 30). The six digits of the source data string "1A23B456CD" are "123456", which encodes to a 20-bit sequence of:

"00011110001001000000"

which is appended to the end of the 13-bit sequence cited at the start of this paragraph.

The four non-digits of the source data string are "ABCD", which are converted (using Table K-1) to a sequence of four Base-30 values 1, 2, 3, and 4 (denoted as values v_3 through v_0 in the formula below. These are then converted to a single Binary value, using the following formula:

$$30^3 * v_3 + 30^2 * v_2 + 30^1 * v_1 + 30^0 * v_0$$

In this example, the formula calculates as (27000 * 1 + 900 * 2 + 30 * 3 + 1 * 4) which is equal to 070DE (hexadecimal) encoded as the 20-bit sequence “00000111000011011110” which is appended to the end of the previous 20-bit sequence. Thus, the AlphaNumeric section contains a total of (13 + 20 + 20) or 53 bits, appended immediately after the previous 83 bits, for a grand total of 136 significant bits in the Packed Object.

The final encoding step is to calculate the full length of the Packed Object (to encode the EBV-6 within the Length Section) and to pad-out the last byte (if necessary). Dividing 136 by eight shows that a total of 17 bytes are required to hold the Packed Object, and that no pad bits are required in the last byte. Thus, the EBV-6 portion of the Length Section is “010001”, where this EBV-6 value indicates 17 bytes in the Object. Following that, the Pad Indicator bit is set to ‘0’ indicating that no padding bits are present in the last data byte.

The complete encoding process may be summarized as follows:

Original input: (17)061031(3912)978123456(10)1A23B456CD

Re-ordered as: (17)061031(10)1A23B456CD(3912)978123456

FORMAT FLAGS SECTION: (empty)

OBJECT INFO SECTION:

ebvObjectLen: 010001

paddingPresent: 0

ebvNumIDs: 001

IDvals: 1111101 0110011

SECONDARY ID SECTION:

IDbits: 0010

AUX FORMAT SECTION:

auxFormatbits: 1 0101

DATA SECTION:

KLnumeric: 0000 11101110 01100111 111010 01001100 11111010 11000000

ANheader: 0

ANprefix: 0

ANsuffix: 0

ANmap: 01 00100011

ANdigitVal: 0001 11100010 01000000

ANnonDigitsVal: 0000 01110000 11011110

Padding: none

Total Bits in Packed Object: 136; when byte aligned: 136

Output as: 44 7E B3 2A 87 73 3F 49 9F 58 01 23 1E 24 00 70 DE

Table L-1 shows the relevant subset of a hypothetical ID Table for a hypothetical ISO-registered Data Format 99.

Table L-1: hypothetical Base ID Table, for representing GS1 Application Identifiers

K-Version = 1.0			
K-TableID = F99B0			
K-RootOID = urn:oid:1.0.15961.9			
K-IDsize = 128			
IDvalue	**OIDs**	**Data Title**	**FormatString**
3	10	BATCH/LOT	1*20an
8	17	USE BY OR EXPIRY	6n
51	391 %x30-39	AMOUNT – 391n	4*18n
125	(17) (10)	EXPIRY + BATCH/LOT	(6n) (1*20an)
K-TableEnd = F99B0			

Appendix M Decoding Packed Objects (non-normative)

M.1 Overview

The decode process begins by decoding the first byte of the memory as a DSFID. If the leading two bits indicate the Packed Objects access method, then the remainder of this Annex applies. From the remainder of the DSFID octet or octets, determine the Data Format, which shall be applied as the default Data Format for all of the Packed Objects in this memory. From the Data Format, determine the default ID Table which shall be used to process the ID Values in each Packed Object.

Typically, the decoder takes a first pass through the initial ID Values list, as described earlier, in order to complete the list of identifiers. If the decoder finds any identifiers of interest in a Packed Object (or if it has been asked to report back all the data strings from a tag's memory), then it will need to record the implied fixed lengths (from the ID table) and the encoded variable lengths (from the Aux Format subsection), in order to parse the Packed Object's compressed data. The decoder, when recording any variable-length bit patterns, must first convert them to variable string lengths per the table (for example, a three-bit pattern may indicate a variable string length in the range of two to nine).

Starting at the first byte-aligned position after the end of the DSFID, parse the remaining memory contents until the end of encoded data, repeating the remainder of this section until a Terminating Pattern is reached.

Determine from the leading bit pattern (see I.4) which one of the following conditions applies:

a) there are no further Packed Objects in Memory (if the leading 8-bit pattern is all zeroes, this indicates the Terminating Pattern)

b) one or more Padding bytes are present. If padding is present, skip the padding bytes, which are as described in Annex I, and examine the first non-pad byte.

c) a Directory Pointer is encoded. If present, record the offset indicated by the following bytes, and then continue examining from the next byte in memory

d) a Format Flags section is present, in which case process this section according to the format described in Annex I

e) a default-format Packed Object begins at this location

If the Packed Object had a Format Flags section, then this section may indicate that the Packed Object is of the ID Map format, otherwise it is of the ID List format. According to the indicated format, parse the Object Information section to determine the Object Length and ID information contained in the Packed Object. See Annex I for the details of the two formats. Regardless of the format, this step results in a known Object length (in bits) and an ordered list of the ID Values encoded in the Packed Object. From the governing ID Table, determine the list of characteristics for each ID (such as the presence and number of Secondary ID bits).

Parse the Secondary ID section of the Object, based on the number of Secondary ID bits invoked by each ID Value in sequence. From this information, create a list of the fully-qualified ID Values (FQIDVs) that are encoded in the Packed Object.

Parse the Aux Format section of the Object, based on the number of Aux Format bits invoked by each FQIDV in sequence.

Parse the Data section of the Packed Object:

a) If one or more of the FQIDVs indicate all-numeric data, then the Packed Object's Data section contains a Known-Length Numeric subsection, wherein the digit strings of these all-numeric items have been encoded as a series of binary quantities. Using the known length of each of these all-numeric data items, parse the correct numbers of bits for each data item, and convert each set of bits to a string of decimal digits.

b) If (after parsing the preceding sections) one or more of the FQIDVs indicate alphanumeric data, then the Packed Object's Data section contains an AlphaNumeric subsection, wherein the character strings of these alphanumeric items have been concatenated and encoded into the structure defined in Annex I. Decode this data using the "Decoding Alphanumeric data" procedure outlined below.

For each FQIDV in the decoded sequence:

a) convert the FQIDV to an OID, by appending the OID string defined in the registered format's ID Table to the root OID string defined in that ID Table (or to the default Root OID, if none is defined in the table)

b) Complete the OID/Value pair by parsing out the next sequence of decoded characters. The length of this sequence is determined directly from the ID Table (if the FQIDV is specified as fixed length) or from a corresponding entry encoded within the Aux Format section.

M.2 Decoding Alphanumeric data

Within the Alphanumeric subsection of a Packed Object, the total number of data characters is not encoded, nor is the bit length of the character map, nor are the bit lengths of the succeeding Binary sections (representing the numeric and non-numeric Binary values). As a result, the decoder must follow a specific procedure in order to correctly parse the AlphaNumeric section.

When decoding the A/N subsection using this procedure, the decoder will first count the number of non-bitmapped values in each base (as indicated by the various Prefix and Suffix Runs), and (from that count) will determine the number of bits required to encoded these numbers of values in these bases. The procedure can then calculate, from the remaining number of bits, the number of explicitly-encoded character map bits. After separately decoding the various binary fields (one field for each base that was used), the decoder "re-interleaves" the decoded ASCII characters in the correct order.

The A/N subsection decoding procedure is as follows:

- Determine the total number of non-pad bits in the Packed Object, as described in section I.8.2
- Keep a count of the total number of bits parsed thus far, as each of the subsections prior to the Alphanumeric subsection is processed
- Parse the initial Header bits of the Alphanumeric subsection, up to but not including the Character Map, and add this number to previous value of TotalBitsParsed.
- Initialize a DigitsCount to the total number of base-10 values indicated by the Prefix and Suffix (which may be zero)
- Initialize an ExtDigitsCount to the total number of base-13 values indicated by the Prefix and Suffix (which may be zero)
- Initialize a NonDigitsCount to the total number of base-30, base 74, or base-256 values indicated by the Prefix and Suffix (which may be zero)
- Initialize an ExtNonDigitsCount to the total number of base-40 or base 84 values indicated by the Prefix and Suffix (which may be zero)
- Calculate Extended-base Bit Counts: Using the tables in Annex K, calculate two numbers:

- ExtDigitBits, the number of bits required to encode the number of base-13 values indicated by ExtDigitsCount, and
- ExtNonDigitBits, the number of bits required to encode the number of base-40 (or base-84) values indicated by ExtNonDigitsCount
- Add ExtDigitBits and ExtNonDigitBits to TotalBitsParsed

- Create a PrefixCharacterMap bit string, a sequence of zero or more quad-base character-map pairs, as indicated by the Prefix bits just parsed. Use quad-base bit pairs defined as follows:
  - ‘00’ indicates a base 10 value;
  - ‘01’ indicates a character encoded in Base 13;
  - ‘10’ indicates the non-numeric base that was selected earlier in the A/N header, and
  - ‘11’ indicates the Extended version of the non-numeric base that was selected earlier
- Create a SuffixCharacterMap bit string, a sequence of zero or more quad-base character-map pairs, as indicated by the Suffix bits just parsed.
- Initialize the FinalCharacterMap bit string and the MainCharacterMap bit string to an empty string
- **Calculate running Bit Counts**: Using the tables in Annex B, calculate two numbers:
  - DigitBits, the number of bits required to encode the number of base-10 values currently indicated by DigitsCount, and
  - NonDigitBits, the number of bits required to encode the number of base-30 (or base 74 or base-256) values currently indicated by NonDigitsCount
- set AlnumBits equal to the sum of DigitBits plus NonDigitBits
- if the sum of TotalBitsParsed and AlnumBits equals the total number of non-pad bits in the Packed Object, then no more bits remain to be parsed from the character map, and so the remaining bit patterns, representing Binary values, are ready to be converted back to extended base values and/or base 10/base 30/base 74/base-256 values (skip to the **Final Decoding** steps below). Otherwise, get the next encoded bit from the encoded Character map, convert the bit to a quad-base bit-pair by converting each ‘0’ to ‘00’ and each ‘1’ to ‘10’, append the pair to the end of the MainCharacterMap bit string, and:
  - If the encoded map bit was ‘0’, increment DigitsCount,
  - Else if ‘1’, increment NonDigitsCount
  - Loop back to the **Calculate running Bit Counts** step above and continue
- **Final Decoding steps:** once the encoded Character Map bits havSe been fully parsed:

- Fetch the next set of zero or more bits, whose length is indicated by ExtDigitBits. Convert this number of bits from Binary values to a series of base 13 values, and store the resulting array of values as ExtDigitVals.
- Fetch the next set of zero or more bits, whose length is indicated by ExtNonDigitBits. Convert this number of bits from Binary values to a series of base 40 or base 84 values (depending on the selection indicated in the A/N Header), and store the resulting array of values as ExtNonDigitVals.
- Fetch the next set of bits, whose length is indicated by DigitBits. Convert this number of bits from Binary values to a series of base 10 values, and store the resulting array of values as DigitVals.
- Fetch the final set of bits, whose length is indicated by NonDigitBits. Convert this number of bits from Binary values to a series of base 30 or base 74 or base 256 values (depending on the value of the first bits of the Alphanumeric subsection), and store the resulting array of values as NonDigitVals.
- Create the FinalCharacterMap bit string by copying to it, in this order, the previously-created PrefixCharacterMap bit string, then the MainCharacterMap string , and finally append the previously-created SuffixCharacterMap bit string to the end of the FinalCharacterMap string.
- Create an interleaved character string, representing the concatenated data strings from all of the non-numeric data strings of the Packed Object, by parsing through the FinalCharacterMap, and:
    - For each ‘00’ bit-pair encountered in the FinalCharacterMap, copy the next value from DigitVals to InterleavedString (add 48 to each value to convert to ASCII);
    - For each ‘01’ bit-pair encountered in the FinalCharacterMap, fetch the next value from ExtDigitVals, and use Table K-2 to convert that value to ASCII (or, if the value is a Base 13 shift, then increment past the next ‘01’ pair in the FinalCharacterMap, and use that Base 13 shift value plus the next Base 13 value from ExtDigitVals to convert the pair of values to ASCII). Store the result to InterleavedString;
    - For each ‘10’ bit-pair encountered in the FinalCharacterMap, get the next character from NonDigitVals, convert its base value to an ASCII value using Annex K, and store the resulting ASCII value into InterleavedString. Fetch and process an additional Base 30 value for every Base 30 Shift values encountered, to create and store a single ASCII character.
    - For each ‘11’ bit-pair encountered in the FinalCharacterMap, get the next character from ExtNonDigitVals, convert its base value to an ASCII value using Annex K, and store the resulting ASCII value into InterleavedString, processing any Shifts as previously described.

Once the full FinalCharacterMap has been parsed, the InterleavedString is completely populated. Starting from the first AlphaNumeric entry on the ID list, copy characters from the InterleavedString to each such entry, ending each copy operation after the number of characters indicated by the corresponding Aux Format length bits, or at the end of the InterleavedString, whichever comes first.

Appendix N Acknowledgement of Contributors and Companies Opted-in during the Creation of this Standard (Informative)

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 TDS 1.5. This list does not acknowledge those who only monitored the process or those who chose not to have their name listed here. Active participants status was granted to those who generated emails, submitted comments during reviews, attended face-to-face meetings, participated in WG ballots, and attended conference calls that were associated with the development of this standard.

Company	Name of Participant	Role
Symbol Technologies Inc, a Motorola Co.	Rick Schuessler	Co-Chair, User Memory Editor
Ahold NV	Vijay Sundhar	Initial Co-Chair
Ken Traub Consulting LLC	Ken Traub	Overall Editor
Impinj, Inc.	Theron Stanford	TID Editor
Kimberly-Clark Corporation	John Anderla	Initial Editor
GS1 EPCglobal, Inc.	Mark Frey	SAG Facilitator, ISO Liaison
GS1 Netherlands	Sylvia Stein	Facilitator
Intermec Technologies Corporation	Sprague Ackley	
GS1 Germany	Craig Alan Repec	
Reva Systems Corporation	Scott Barvick	

GS1 Spain	Sergi Cardona	
Lockheed Martin	Denton Clark	
NXP Semiconductors	Henk Dannenberg	
GS1 Canada	Kevin Dean	
GS1 US	Ray Delnicki	
Target Corporation	Roberto DeVet	
Procter & Gamble Company	John Duker	
Nestle S. A.	Vera Feuerstein	
Reva Systems Corporation	Jeff Fischer	
7 id (Formerly EOSS)	Gerhard Gangl	
GS1 Switzerland	Heinz Graf	
GS1 Global Office	Scott Gray	
Michelin	Marc Hammer	
Q.E.D. Systems	Craig Harmon	
Intermec Technologies Corporation	Barba Hickman	
GS1 US	Bernie Hogan	
Supply Insight, Inc.	Robert Hotaling	
Paxar	John Kessler	
Michelin	Pat King	
Track Tracer AS	Steinar Kjærnsrød	
Metro	Bastian Konigs	
GS1 Hungary	Peter Kurucz	
Sandlab Corp.	Kay Labinsky	
Goodyear Tire & Rubber Company	Steve Lederer	
Auto-ID Labs - Cambridge	Dr. Mark Harrison	
Pfizer, Inc.	Tim Marsh	
Away (formerly Cyclone)	Dale Moberg	
WalMart Stores, Inc.	Ron Moser	
Kraft Foods, Inc.	Doug Naal	
NXP Semiconductors	Elista Polizveva	
Cisc Semiconductor	Josef Preishuber-Pfluegl	
France Telecom Orange	Matt Robshaw	
Supply Insight, Inc.	Felix Rodriguez	

Intelleflex Corporation	Lauren Schlicht	
GS1 Australia	Sue Schmid	
INRIA	Loic Schmidt	
GS1 Austria GmbH	Eugen Sehorz	
Lockheed Martin	Ryad Semichi	
GS1 Japan	Yuko Shimizu	
Department of Homeland Security	Rajiv Singh	
Schering-Plough Corp.	Mike Smith	
EPCglobal North America	Michele Southall	
EM Microelectronic Marin Sa	Jim Springer	
Target Corporation	Joe Spritzer	
GS1 Hong Kong	KK Suen	
CHEP International, Inc	Jim Sykes	
Intermec Technologies Corporation	Phyllis Turner-Brim	
GS1 China	Yi Wang	
GS1 UK	David Weatherby	
GS1 France	Lionel Willig	
Auto-ID Labs-ICU	Jong Woo Sung	
GS1 China	Ruoyan Yan	
Supply Insight, Inc.	Graham Yarbrough	

The following list in alphabetical order contains all companies that were opted-in to the Tag Data and Translation Standard Working Group and have signed the EPCglobal IP Policy as of March 24, 2010.

Company Name
7iD (formerly EOSS GmbH)
Afilias Limited
AFNIC
Ahold NV
Allixon Co., Ltd
AMOS Technologies Inc.
AT4 Wireless (formerly Cetecom Spain)
Atmel GmBH
Auto-ID Labs - Adelaide
Auto-ID Labs - Cambridge
Auto-ID Labs - Fudan University
Auto-ID Labs - ICU
Auto-ID Labs - Japan
Auto-ID Labs - MIT
Auto-ID Labs - University of St Gallen
AXWAY/formerly Cyclone
Benedicta
Chang Jung Christian University Rfid Research Center
Cheng-Loong Corporation
CHEP International, Inc
Cisc Semiconductor Design and Consulting Gmbh
Department of Homeland Security
DHL Logistics GmbH (Deutsche Post)
ECO, Inc.
EM Microelectronic Marin Sa
EPCglobal Inc.
ETRI - Electronics & Telecommunication Research Institute
Evtek Univerisity Of Applied Science
Feng Chia University, Department of Information Engineering and Computer Science
France Telecom Orange
Fujitsu Limited

Glaxo Smith Kline

Goodyear Tire & Rubber Company

GS1 Australia

GS1 Austria GmbH

GS1 Brasil

GS1 Canada

GS1 China

GS1 EPCglobal, Inc.

GS1 France

GS1 Germany

GS1 Global Office

GS1 Hong Kong

GS1 Hungary

GS1 Japan

GS1 Netherlands

GS1 South Korea

GS1 Spain

GS1 Sweden AB

GS1 Switzerland

GS1 Taiwan

GS1 UK

GS1 US

iControl, Inc.

Impinj, Inc.

Innovision Research & Technology plc

INRIA

Intelleflex Corporation

Intermec Technologies Corporation

Johnson & Johnson

Ken Traub Consulting LLC

Kimberly-Clark Corporation

KL-NET (Korea Logistics Network Corp.)

Kraft Foods, Inc.

LIT (Institute of Logistics Information Technology)

Lockheed Martin - Savi Technology Division

Lockheed Martin, Corp.

MetaBiz

METRO AG

Michelin

Microelectronics Technology, Inc.

MITSUI & CO., LTD.

MOE RFID Project Office

National Taiwan University of Science & Technology, college of electrical Engine

NEC Corporation

Nestle S. A.

NXP Semiconductors

Packaging Corporation of America

Paxar

Pfizer, Inc.

PowerID Ltd

Printronix

Procter & Gamble Company

Q.E.D. Systems

RetailTech

Sandlab Corp.

Schering-Plough Corp.

Supply Insight, Inc.

Symbol Technologies Inc, a Motorola Co.

Target Corporation

The Boeing Company

Tibco Software, Inc

Toppan Printing Co., Ltd

TraceTracker AS

Tranz Technologies Inc

VeriSign

WalMart Stores, Inc.

Yuen Foong Yu Paper (Yeon Tech)

Zebra Technologies Corporation