INTERNATIONAL STANDARD

ISO/IEC 15962

First edition
2004-10-15

Information technology — Radio frequency identification (RFID) for item management — Data protocol: data encoding rules and logical memory functions

Technologies de l'information — Identification par radiofréquence (RFID) pour la gestion d'objets — Protocole de données: règles d'encodage des données et fonctions logiques de mémoire

Reference number
ISO/IEC 15962:2004(E)

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 15962 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 31, *Automatic identification and data capture techniques*.

Introduction

The technology of Radio Frequency Identification (RFID) is based on non-contact electronic communication across an air interface. The structure of the bits stored on the memory of the RF tag is invisible and accessible between the RF tag and the interrogator only by the use of the appropriate air interface protocol, as specified in the different ISO/IEC 18000 parts. The transfer of data between the application and the interrogator in open systems requires data to be presented in a consistent manner on any RF tag that is part of that open system. Functional commands from the application and responses from the interrogator also require being processed in a standard way. This is not only to allow equipment to be interoperable, but in the special case of data carrier, for the data to be encoded on the RF tag in one systems implementation for it to be read at a later time in a completely different and unknown systems implementation. The data bits stored on each RF tag must be formatted in such a way as to be reliably read at the point of use if the RF tag is to fulfil its basic objective. The integrity of this is achieved through the use of a data protocol as specified in ISO/IEC 15961 and this International Standard.

Manufacturers of radio frequency identification equipment (interrogators, RF tags, etc) and the users of RFID technology require a publicly available data protocol for RFID for item management. ISO/IEC 15961 and this International Standard specify this data protocol, which is independent of any of the air interface standards defined in the various parts of ISO/IEC 18000. As such, the data protocol is a consistent component in the RFID system that may independently evolve to include additional air interface protocols

The transfer of data to and from the application, supported by appropriate commands is the subject of the companion standard: ISO/IEC 15961. This International Standard specifies the overall process and the methodologies developed to format the application data into a structure to store on the RF tag.

Information technology — Radio frequency identification (RFID) for item management — Data protocol: data encoding rules and logical memory functions

1 Scope

The data protocol used to exchange information in an RFID system for item management is specified in ISO/IEC 15961 and in this International Standard. Both International Standards are required for a complete understanding of the data protocol in its entirety; but each focuses on one particular interface:

- ISO/IEC 15961 addresses the interface with the application system.
- This International Standard deals with the processing of data and its presentation to the RF tag, and the initial processing of data captured from the RF tag.

This International Standard focuses on encoding the transfer syntax, as defined in ISO/IEC 15961 according to the application commands defined in that International Standard. The encodation is in a Logical Memory as a software analogue of the physical memory of the RF tag being addressed by the interrogator.

This International Standard

- defines the encoded structure of object identifiers;
- specifies the data compaction rules that apply to the encoded data;
- specifies a Precursor for encoding syntax features efficiently;
- specifies formatting rules for the data, e.g. depending on whether a directory is used or not;
- defines how application commands, e.g. to lock data, are transferred to the Tag Driver;
- defines other communication to the application.

NOTE Conventionally in International Standards, long numbers are separated by a space character as a "thousands separator". This convention has not been followed in this International Standard, because the arcs of an object identifier are defined by a space separator (according to ISO/IEC 8824 and ISO/IEC 8825). As the correct representation of these arcs is vital to this International Standard, all numeric values have no space separators except to denote a node between two arcs of an object identifier.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 8824-1, *Information technology — Abstract Syntax Notation One (ASN.1) — Specification of basic notation* (equivalent to ITU-T Recommendation X.680)

ISO/IEC 8825-1, *Information technology — ASN.1 encoding rules — Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)* (equivalent to ITU-T Recommendation X.690)

ISO/IEC 15961:2004, *Information technology — Radio frequency identification (RFID) for item management — Data protocol: application interface*

ISO/IEC 18000 (all parts), *Information technology — Radio frequency identification for item management*

ISO/IEC 19762-1, *Information technology — Automatic identification and data capture (AIDC) techniques — Harmonized vocabulary — Part 1: General terms for AIDC*[1)]

ISO/IEC 19762-3, *Information technology — Automatic identification and data capture (AIDC) techniques — Harmonized vocabulary — Part 3: Radio frequency identification (RFID)*[1)]

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 19762-1, ISO/IEC 19762-3 and the following apply.

NOTE For terms defined below and in ISO/IEC 19762-1 or ISO/IEC 19762-3, the definitions given below apply.

3.1.1 Application commands
The instruction issued from the application to the Data Protocol Processor in order to initiate an action or operation with the RF tag(s) via the interrogator.

3.1.2 Application memory
The area of the RF tag available for storing data written to it. Sometimes known as *user memory*.

3.1.3 Arc
A specific branch of an object identifier tree, with new arcs added as required to define a particular object. The top three arcs of all object identifiers compliant with ISO/IEC 9834-1 are defined in Annex A of ISO/IEC 15961.

3.1.4 Block
The minimum number of bytes on an RF tag that can be in a write transaction, or read transaction, across the air interface.

3.1.5 Command / Response Unit
That part of the Data Protocol Processor that processes application commands and sends responses to control encoding, decoding, structuring of the Logical Memory and transfer to the Tag Driver.

---

1) To be published.

3.1.6 Data compaction
A mechanism, or algorithm, to process the original data so that it is represented efficiently in fewer bytes in a data carrier than in the original presentation.

3.1.7 Data Compactor
The implementation of the data compaction process defined in this International Standard.

3.1.8 Data Protocol Processor
The implementation of the processes defined in this International Standard, including the Data Compactor, Formatter, Logical Memory, and Command/Response Unit.

3.1.9 elementName
A component of a ReferenceType or enumerated list in ASN.1 Syntax.

3.1.10 Formatter
The implementation of the data formatting process defined in this International Standard.

3.1.11 Logical Memory
A software analogue on the Data Protocol Processor of the Logical Memory Map.

3.1.12 Logical Memory Map
An array of contiguous bytes of memory on the RF tag, representing the application (or user) memory to be used exclusively for the encoding of objects, objectIds, and their associated Precursor on the RF tag. The system information shall be defined by different means or stored in a separate area on the RF tag. This can be achieved by partitioning memory, partly for system information and mainly for the Logical Memory Map purpose.

3.1.13 Object
A well-defined piece of information, definition, or specification which requires a name in order to identify its use in an instance of communication.

3.1.14 Object identifier
A value (distinguishable from all other such values) which is associated with an object.

3.1.15 OBJECT IDENTIFIER type
A simple ASN.1 type whose distinguished values are the set of all object identifiers allocated in accordance with the rules of ISO/IEC 8824-1 (ITU-T X.680).

3.1.16 Precursor
A byte, sometimes a sequence of bytes, within the encodation on the Logical Memory and Logical Memory Map that acts as metadata for the subsequent objectId and object.

3.1.17 RELATIVE-OID type
A particular object identifier where a common root-OID (for the first and subsequent arcs) is implied, and remaining arcs after the root-OID are defined by the RELATIVE-OID.

3.1.18 Response
The feedback received by the application from an application command sent to the Data Protocol Processor.

3.1.19 root-OID
That part of a set of OBJECT IDENTIFIERS that has a common first, second, and subsequent arcs. The root-OID is a prefix to a RELATIVE-OID to construct a complete OBJECT IDENTIFIER.

3.1.20 Tag Driver
The implementation of the process to transfer data between the Data Protocol Processor and the RF tag.

3.1.21 Transfer syntax
The abstract syntax and concrete syntax used in the transfer of data between open systems.

NOTE: The term "transfer syntax" is sometimes used to mean encoding rules, and sometimes used to mean the representation of bits in data while in transit.

3.2 Abbreviated terms

BER	Basic Encoding Rules (of ASN.1)
EAN.UCC	EAN International & Uniform Code Council, Inc
IATA	International Air Transport Association
UPU	Universal Postal Union

4 Protocol model

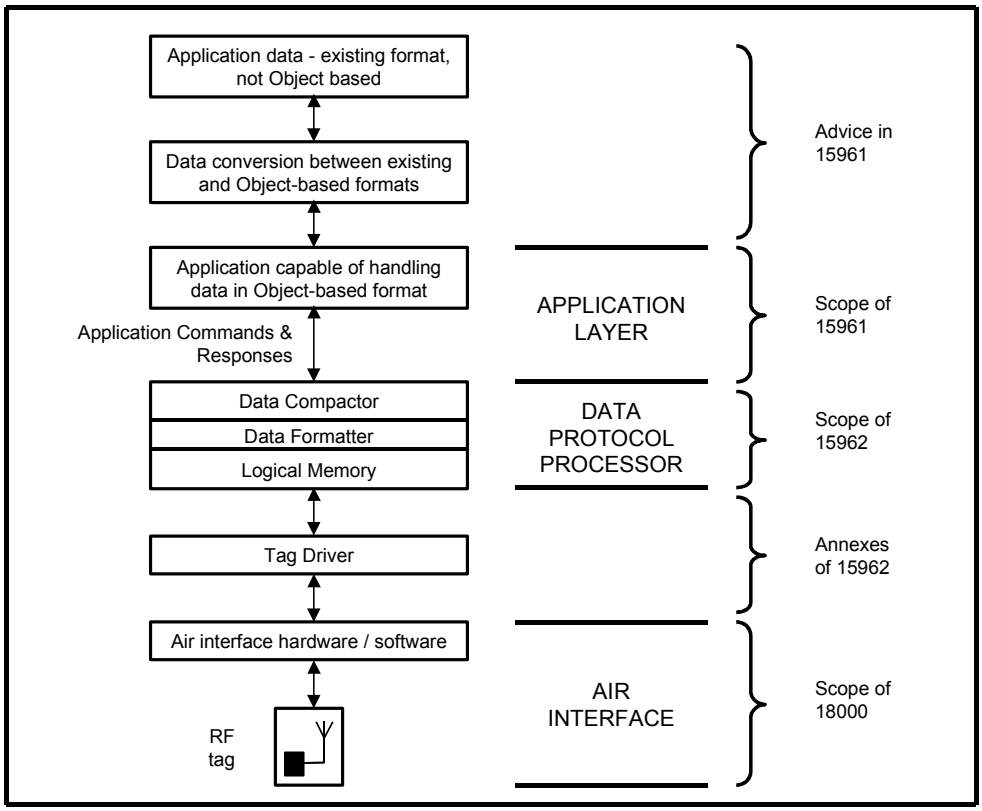


Figure 1 — Schematic of Protocol Layers for an Implementation of RFID for Item Management

4.1 Overview

RFID supports bit encodation in the RF tag memory. Unlike other data carrier standards prepared by ISO/IEC JTC1 SC31 which require encodation schemes that are specific to the individual data carrier technology, ISO/IEC 18000 does not specify the interpretation of bits or bytes encoded on the RF tag memory. However, as an RF tag is a relay in a communication system, each tag used for open systems item management needs to have data encoded in a consistent manner. The prime function of ISO/IEC 15961 is to specify a common interface between the application programs and the RF interrogator. The prime function of this International Standard is to specify the common encoding rules and logical memory functions.

RF tags utilise electronic memory that is typically capable of increasing data capacity as new generations of product are introduced. Differences in data capacity of each RF tag type, whether similar or dissimilar, are recognised by the data protocol defined in these two International Standards.

Different application standards may have their own particular data sets or data dictionaries. Each major application standard for item management needs to have its data treated in an unambiguous manner, avoiding confusion with data from other applications and even with data from closed systems. The data protocol specified in these International Standards ensures the unambiguous identification of data.

4.2 Layered protocol

The protocol layers of an implementation of RFID for item management are illustrated schematically in Figure 1.

The data protocol specified in this International Standard is independent of the different RF tag technologies specified in ISO/IEC 18000, which is concerned with different air interface protocols that function between the interrogator and the RF tag. This independence is achieved by implementing the standards at different levels in the protocol hierarchy. The RFID data protocol defined in this International Standard is primarily concerned with the upper layers as described below:

Application layer - as defined in ISO/IEC 15961

- The RFID data protocol specifies how data is presented as objects, each uniquely identified with an object identifier, which are meaningful to the application and can be encoded on the RF tag.
- This RFID data protocol defines application commands and responses so that application programs can specify what data to transfer to and from the RF tag and to append, update or delete data on the RF tag.
- This RFID data protocol also defines error messages as responses to the application.

The application interface of this RFID data protocol is based on ASN.1, which:

- provides a means of defining the protocol which is independent of the host application, operating system, and programming language and also independent of the specific command structures between the interrogator and tag driver.
- identifies any data object distinctly from all others using object identifiers, even to enable different data formats to be intermixed on the same RF tag.
- defines unambiguous commands and responses, so that they can be intermixed with data on the same wired or wireless network.
- provides the abstract syntax for defining the commands and responses in a structured and consistent and verifiable manner, and provides the transfer syntax that defines the byte stream transferred between the processes of ISO/IEC 15961 and those of this International Standard.
- enables implementation in a variety of computer languages through the use of compilers, alternatively programs can be written from the specification. In either case there is a vital need for the transfer syntax to be fully consistent and compliant to function in open systems where the sender and recipient can be unknown to one another.

Data Protocol Processing - as defined in this International Standard

- The RFID data protocol specifies how data is encoded, compacted and formatted on the RF tag and how this data is retrieved from the RF tag to be meaningful to the application.
- This RFID data protocol provides for a set of schemes that compact the data to make more use of the memory space.
- This RFID data protocol also supports various storage formats to enable efficient use of memory and efficient access procedures.

All these features are described and specified later in this International Standard and its companion standard. Figure 1, and the outline description above, applies to a general process. Different rules may apply to RF tags that are capable of executing commands (see 7.3.3).

This RFID data protocol specifies the application level communication and the RF tag interrogator level rules for data encoding, compaction and storage formats. This protocol may be implemented:

- on the same platform as the application.
- on a separate platform linked to the application platform e.g. linked via a serial link, LAN or internet connection.
- on an embedded platform e.g. in a bar code printer/RFID encoder, in a bar code/RFID scanner, or a dedicated RFID interrogator.

This RFID data protocol has been designed such that the actual platform on which it is implemented is transparent to the application. It is also independent of the programming language used by the application. If both standards are not implemented, care will need to be taken to maintain the functionality between the two standards. The compliance clauses of both standards address these points in greater detail.

The rules specified in these International Standards create a complete independence between the application and the technology of the air interface and RF tag. The type of RF tag used in an implementation can be changed without requiring the application to change.

4.3 Functional processes

There are various functional processes that need to take place to write data to an RF tag and to read data from it. Figure 2 shows a schematic of an implementation where the processing of the data protocol resides in the interrogator. This illustration is provided to help with the understanding of the processes, and although a typical implementation, many others are possibly compliant with this data protocol.

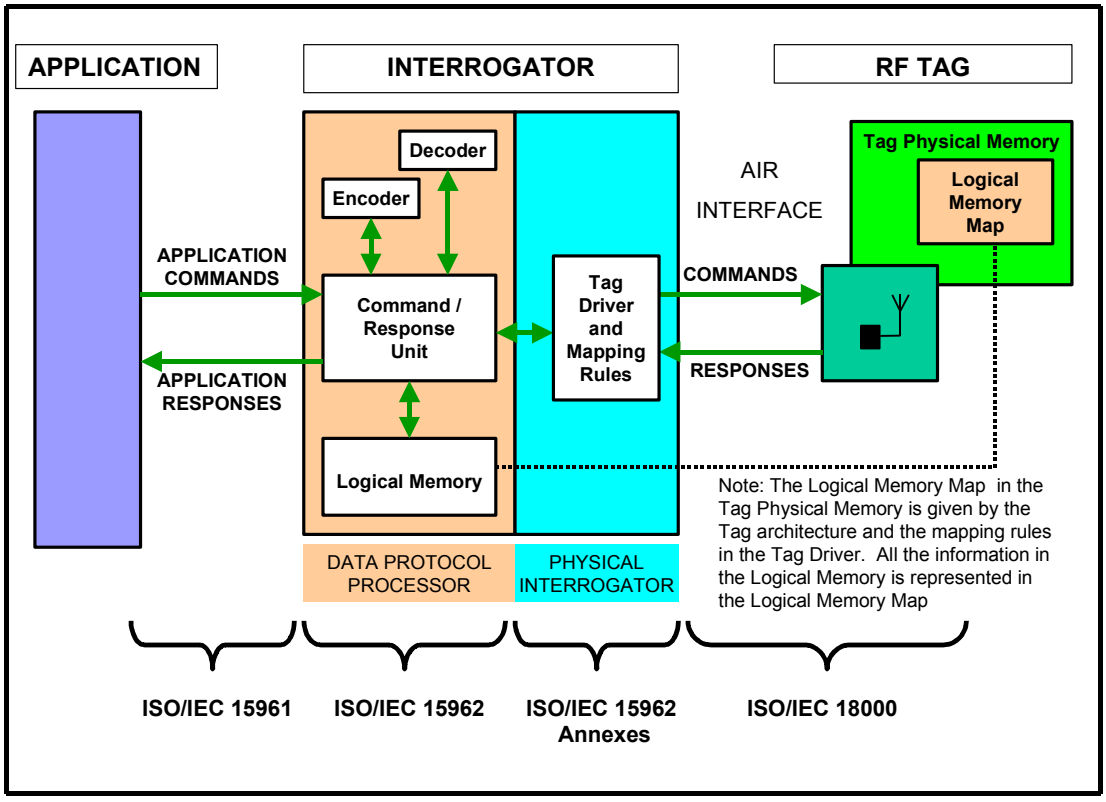


Figure 2 — Logical Functions and Interfaces

Application is the user application database and software.

The data flows between the application and the Data Protocol Processor are formatted according to ISO/IEC 15961 and are uncompacted. However, there are numerous established systems where data is formatted to be compliant, for example, with a bar code related syntax. It is therefore reasonable to insert interface modules in the data flow to convert from and to existing application formats.

NOTE: Careful consideration should be given to the extent that established systems need to be supported relative to the potential benefits to be gained from adopting the data protocol specified in ISO/IEC 15961 and this International Standard. This is because this protocol has been developed around the features of RFID, such as selective read/write and the ability to lock data. Older protocols are unlikely to support such features.

Interrogator is the module in which all the basic processing of the data protocol takes place and there is an interface to the RF tag.

Data Protocol Processor provides all the processing, which is as specified in this International Standard and is required for handling application data. It consists of the following components, all of which are described more fully below: Command/Response Unit, Logical Memory, Encoder (which supports a Data Compactor and Formatter function) and Decoder (which supports the inverse functions of the Encoder). The Data Protocol Processor can physically reside anywhere between the application software and the tag driver but shall contain all the components.

Command/Response Unit for receiving the application commands from the application in a format specified in ISO/IEC 15961, acting upon these commands where appropriate and converting to the specific RF tag lower level command codes.

EXAMPLE:

An application command of *write Data Object {name}* is application related. The data protocol recognises this and can format the data onto the Logical Memory in the Data Protocol Processor. The information from the particular RF tag is required to set the parameters of the Logical Memory Map (e.g. number of octets, whether a directory is in use, etc) on the RF Tag. The Tag Driver converts the application command into a tag-specific command.

It can be seen from this example that there is a distinct boundary between the Data Protocol Processor and the Tag Driver.

Logical Memory. This is an array of contiguous octets (or bytes) of memory acting as a common representation of the Logical Memory Map in the user memory of the RF tag to which the object identifiers and data objects are mapped in octets. The Logical Memory takes into account some parameters of the real RF tag, for example the block size, the number of blocks and the storage format. The Logical Memory ignores any detailed tag architecture.

The use of the Logical Memory means that an application can interface with an application-compliant RF tag, but that individual RF tags can have completely different memory capacities and architectures. This enables an implementation to benefit from new technological developments permitted within the framework of ISO/IEC 18000, such as larger capacity or faster access RF tags, without changing the application.

Encoder controls the process of writing data through the functional processes performed by the Data Compactor Module and Formatter Module.

Data Compactor provides the standard compaction rules to reduce the number of octets stored on the RF tag and transferred across the air interface. Numeric data, for example, is octet based to some coded character set for the application, but can be encoded in a compact form on the RF tag memory.

Formatter provides the processes to place the object identifier and object (data) into an appropriate and efficient format for storing on the Logical Memory.

NOTE: The physical mapping of bits to comply with the RF tag architecture is performed by the Tag Driver.

Decoder controls the process of reading and interpreting data through the functional processes performed by the Data De-compactor Module and De-formatter Module.

Tag Driver provides two main functions:

- It maps the contents of the Logical Memory to the Logical Memory Map of the RF tag in use.
- It provides facilities that accept the application commands of this data protocol, and converts them to a format that results in calls to command codes supported by the particular RF Tag. For example, an application command **write Data Object {name}** could result in the RF tag related command of write (block #, data).

The description of the tag driver for particular RF tags is provided in annexes of this International Standard. For the purpose of this International Standard, a tag driver is unique to a particular air interface type of RF tag as specified in the appropriate part of ISO/IEC 18000. This is a logical representation; physical implementation could combine features of different logical tag drivers. An interrogator may support one or many tag drivers.

RF Tag, although beyond the scope of these International Standards, is shown to complete the flow of data and commands.

Logical Memory Map represents all the data in the Logical Memory of the Data Protocol Processor converted (or mapped) to a location structure determined by the mapping rules in the Tag Driver and the architecture of the RF tag.

5 Data structure

5.1.1 The 8-bit byte

This International Standard supports the encoding of data and data objects aligned as 8-bit bytes (referred to as octets in ISO/IEC 15961 to be more compatible with communications standards). Within each 8-bit byte (hereafter 'byte'), the most significant bit is bit 8 and the least significant is bit 1. Accordingly, the weight allocated to each bit is:

Bit Value	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1
Weight	128	64	32	16	8	4	2	1

5.1.2 N-bit encoding

This International Standard supports the encoding of bit fields that are not necessarily aligned on a byte boundary. The most significant bit is bit N and the least significant bit is bit 1. When a sequence of bit fields is defined, the first bit field occupies the most significant bits.

6 Data Protocol Processor and the application interface

The application shall provide transfers of commands, which may contain embedded data, to the Data Protocol Processor in the format that is compliant with the transfer syntax specified in ISO/IEC 15961.

NOTE: For the remainder of this International Standard, it will be assumed that the transfer mechanism from the application is fully compliant with the Basic Encoding Rules used in ISO/IEC 15961. Implementations that do not use all the specifications of ISO/IEC 15961 need to ensure that transfers are achieved in a completely reliable manner.

The following subclauses describe, in an overview, how the transfer syntax is presented and processed.

6.1 Processing transfers from ISO/IEC 15961

The transfers across the application interface shall be compliant with the transfer syntax of ISO/IEC 15961. This results in an octet aligned stream that intermixes commands and data on the transfer from the application to the Data Protocol Processor, and intermixes response codes and data in communications in the reverse direction. These transfers require a precisely defined syntax that basically has a structure of three components of a Type, Length and Value (or Content), sometimes known as a **TLV** format.

Two formats are permitted:

- Primitive encoding, that is a simple sequence of TLV elements
- Constructed encoding (e.g. TL, TLV, TLV...), where the first V is replaced by triplets of TLV elements

The Data Protocol of ISO/IEC 15961 and this International Standard uses the simpler Primitive encoding, except where the rules of ISO/IEC 8825-1 require that constructed encoding shall be used. Further details are given in 6.2.1 of ISO/IEC 15961.

The **Type** identifies a code that uniquely distinguishes the context of the associated value. The structure of the Type code is defined in 6.2.2 of ISO/IEC 15961 and has a length of one byte.

The **Length** identifies the length in octets (bytes) of the associated value. The rules for length encoding are defined in 6.2.3 of ISO/IEC 15961.

All the **Value** (contents) used in the Data Protocol standards of ISO/IEC 15961 and this International Standard have been specified to reduce to a Universal Type defined in ISO/IEC 8824-1. These are fully defined in subsequent clauses of this International Standard where their significance is relevant.

6.2 Universal Types

The transfer syntax supports a number of universal types that are fundamental to the syntax; sometimes called "built-in types". Each has been assigned a class tag in ISO/IEC 8824-1 to unambiguously identify each type of data. Universal types are shown in capital (uppercase) letters e.g. **UNIVERSAL**. The Universal Types used in this International Standard, together with their Class Tags, are shown in Table 1. This also shows whether Primitive or Constructed encoding shall be used with this Data Protocol and the resultant Type code (as defined in 6.2.2 of ISO/IEC 15961.

Table 1 — Universal Types Used in this Standard

Universal Type	Class Tag	Primitive/ Constructed	Type Code (binary)
BOOLEAN	1	Primitive	00000001
INTEGER	2	Primitive	00000010
OBJECT IDENTIFIER	6	Primitive	00000110
OCTET STRING	4	Primitive	00000100
RELATIVE-OID (reserved for future commands)	13	Primitive	00001101
SEQUENCE & SEQUENCE OF	16	Constructed	00110000

As the Type code is at the beginning of each TLV sequence, it is possible to use the encoded value to determine, in conjunction with the command and response syntax, how to process the associated value.

6.3 Length encoding and decoding

The rules (defined in Clause 6.2.3 of ISO/IEC 15961) encode the length field (n) into an octet-aligned-bit-field. The rules that follow are an inverse of the ISO/IEC 15961 rules, but based on the byte values of the resultant encoding. Typically one or more bytes defines the length of the following value element.

a. If the lead byte value is in the range 01_{HEX} to $7F_{HEX}$, the length is encoded in this single byte. Convert to a bit string, 0 [*lllllll*], strip off the leading zero bit and read the length as a binary value from the remaining 7 bits. The length of the value is <128.

b. If the lead byte value is in the range 80_{HEX} to FF_{HEX}, the length is encoded in this and subsequent byte(s), and is decoded using the following procedure:

   1. Convert the lead byte to a bit string, 1 [nnnnnnn], strip off the leading 1 bit and read the number of subsequent length bytes as a binary value from the remaining 7 bits. The number of subsequent bytes is <127, because the lead byte 01111111_{BIN} is not permitted by the current rules of ISO/IEC 8825-1.

      NOTE: It is extremely unlikely that the value nnnnnnn will be large with respect to this Data Protocol for RFID for item management.

   2. Count the number of bytes from step b1 to determine the bytes that define the length encoding.

   3. Convert this byte string as a binary value of the length of the associated value.

      EXAMPLE:

      This example is of a length ≥128 bytes

Length bytes	82 01 65_{HEX}
Step b1	82_{HEX} = 10000010
	0000010 indicates that there are 2 bytes to determine length
Step b2	01 65_{HEX} determines the length
Step b3	01 65_{HEX} = 1 01100101 = 357 bytes = length of value

6.4 Decoding of Type values

The value (or content) component shall have its Universal Class determined by its preceding Type code; and its length determined by its preceding length bytes. The Type value can then be decoded using the specific rule for the Type, as defined in the following subclauses. This is the first stage of processing and encoding the application data into the Logical Memory. A number of the Type values are not encoded, for example; command codes and arguments; while others are subject to further processing before being encoded more efficiently in the Logical Memory. The sub-clauses that follow only address the decode rules.

6.4.1 Decoding of a BOOLEAN value

The Boolean value is encoded in a single byte. If the value of this is 00_{HEX}, the Boolean value is FALSE. All other values convert to a Boolean value of TRUE.

6.4.2 Decoding an INTEGER value

The Integer value is encoded in one or more bytes, as determined by the preceding length component.

If bit 8 of the lead byte of the Integer value has the value 0, the Integer value is positive. Strip out the lead bit and convert the remaining bit string as the Integer value. To avoid the possibility of bit 8 of the lead byte being 1, pad octets may be introduced into the encoding process of ISO/IEC 15961. These extend the length of the encoded Integer value but do not affect the decode process.

EXAMPLES:

01000101		= 69	
00000000	10000000	= 128	The pad byte is present to avoid the first bit being 1, which would wrongly signal a negative integer
00000001	00000000	= 256	
01111111	11111111	= 32767	

If bit 8 of the lead byte of the Integer value has the value 1, the Integer value is negative. Conversion takes place on the remaining bits with the least significant bit being in position 0. The first stage is to create a decimal integer value as the sum of the values of 2^n. The second stage takes this as a positive decimal integer from which is subtracted the value 2^p, where p is the position number of the lead bit that identifies this as a negative integer. As an equation, this is:

$$\sum_{n=0}^{p-1} 2^n - 2^p$$

EXAMPLE:

10010110	01000110		
1			indicates -ve
0010110	01000110	=	5702
$2^p = 2^{15} =$			32768
5702 - 32768 = -27066			

6.4.3 Decoding an OBJECT IDENTIFIER value

The Object Identifier value is encoded over 2 or more bytes. No decode process is invoked, but there are specific conversion rules that apply to encoding this value onto the Logical Memory, using one of the following two options:

A. The Object Identifier is separated into two constituent parts of a root-OID and a RELATIVE-OID, based on the dataFormat (see 8.3).

B. If Option A is not possible to invoke, it is encoded as a full OBJECT IDENTIFIER. In this case, the OBJECT IDENTIFIER is encoded as presented by the byte string of the transfer encoding from the application.

6.4.4 Decoding the OCTET STRING value

The Octet String value is encoded over one or more bytes. No decode process is invoked, but there are specific compaction rules (see 8.2) that apply to encoding this value onto the Logical Memory.

6.4.5 Decoding the SEQUENCE or SEQUENCE OF value

For decode purposes, Sequence and Sequence Of are treated in a similar manner. The first Length component (shown as L_S below) of the TLV structure determines the length of the complete Sequence (or Sequence Of).

T L_S {TLV TLV TLV ...}

Therefore, it is advisable, in the decode process, to keep a reducing count of the number of bytes still remaining.

The first byte after the Length component L_S is the Type code of the first TLV triplet in the Sequence, or Sequence Of. The next byte(s) are the Length component of the first TLV triplet. Decoding this to the rules defined in 6.3 yields the length of the Value component of the first TLV triplet. The Value component,

identified by its Type and Length is decoded by the relevant process defined in 6.4.1 to 6.4.4. The process is repeated until the number of bytes following, and defined by, the Length component L_S of the Sequence, or Sequence Of, value has been processed.

EXAMPLE:

Transfer encoding:

- 30 14 04 07 {ABC1234} 04 06 {widget} 02 01 $0C_{HEX}$
  NOTE: Values in { } are literal printable characters, presented in this way for ease of illustration. In a real transfer these would be encoded as a string of byte values.
- 30 = SEQUENCE or SEQUENCE OF as per Table 1
- 14 = length L_S = 20 bytes of the complete sequence that follows
- 04 = T of TLV triplet = OCTET STRING
- 07 = L of TLV triplet = 7 bytes
- ABC1234 = literal translation of the 7 bytes of the OCTET STRING value
- At this stage, 9 bytes of the overall length L_S have been processed, with 11 bytes remaining
- 04 = T of next TLV triplet = OCTET STRING
- 06 = L of TLV triplet = 6 bytes
- widget = literal translation of the 6 bytes of the OCTET STRING value
- At this stage, a further 8 bytes of the overall length L_S have been processed, with 3 bytes remaining
- 02 = T of next TLV triplet = INTEGER
- 01 = L of TLV triplet = 1 byte
- 0C = 12

7 Data Protocol Processor and the air interface

The Data Protocol Processor receives communications across the air interface, via the Tag Driver. This clause defines the basic requirements that enable the Data Protocol Processor and RF tag to transfer and share information.

7.1 Air interface services

This International standard is open-ended with respect to the fact that new types of RF tag may be added to the ISO/IEC 18000 multi-part standard and leave the Data Protocol Processor unaltered. To achieve this, some basic presumptions are made about the types of RF tag in ISO/IEC 18000.

- Application memory is an integer number of bytes.

- Application memory shall be organised in blocks. These shall be fixed size and be of one or more bytes.

- The individual blocks shall each be accessible by read and/or write.

  NOTE: This applies to the basic function, additional features may be used to restrict access to authorised users.

- In addition to the requirements (above) relating to the memory, there shall be a reliable mechanism for writing and reading to and from the application memory.

Each type of RF tag in ISO/IEC 18000 shall provide the following air interface services of the physical characteristics and data capabilities of the RF tag:

1. Provide a byte location addressing mechanism from the beginning to the end of the application memory, starting from byte 0. This shall map to the Logical Memory.

2. Provide a mechanism to address a specific RF tag using a permanent or virtual tag identifier.

3. Provide the block size (in bytes).

4. Provide the value of the number of blocks in the application memory.

5. Optionally support selection RF tag inventory and/or addressing of groups of RF tags according to the Application Family Identification mechanism.

6. Provide a mechanism to write and read the storage format.

7. Return a positive or negative acknowledgement (error code) to read and to write commands.

8. Optionally provide the writing capabilities supported by the RF tag.

9. Optionally provide the locking capabilities supported by the RF tag.

10. Support the ability to query the status (locked or not locked) of the memory blocks.

These air interface services should be provided by the Tag Driver. A description of a generic Tag Driver is provided in Annex A. Details of specific Tag Drivers are provided in Annex B.

7.2 Defining the system information

The systems information is a set of elements that is encoded, or provided by other means, from the RF tag to the Data Protocol Processor when communications are established. The systems information shall consist of the following elements and shall be as defined in the following subclasses:

tagId	up to 256 bytes
physical block size	1 byte
number of blocks	1 or more bytes
applicationFamilyId, which itself consists of:	
applicationFamily	4 bits
applicationSubFamily	4 bits
storageFormat, which itself consists of:	
accessMethod	2 bits
reserved for future use	1 bit
dataFormat	5 bits

NOTE: Those elements shown in bold are in the style of the elementNames of ISO/IEC 15961, and are transferred across the application interface by the commands and responses. This style is used in the remainder of this International Standard.

7.2.1 Tag Identifier

The Tag Identifier (**TagId**) is the means of ensuring reliable communication between the application and the RF tag throughout the transaction process via the Data Protocol Processor. The Tag Identifier shall be up to 256 bytes long. It is communicated across the air interface and the application interface. The format of the Tag Identifier shall be as defined for each Tag Driver (see Annex B) and shall be based on one of the following:

a. A completely unique ID programmed in the RF tag, as specified in the ISO/IEC 18000 series).

b. A data related identifier (e.g. like the unique identifier of transport units as specified in ISO 15459-1), that provides for uniqueness within the particular domain of item management or logistics. This requires the data to be read to establish the Tag Id.

c. A virtual or session ID based on a time slot or other feature managed by the air interface protocol.

d. Combinations of (b) and (c), e.g. a virtual ID across the air interface, but requiring the data related identifier to be returned as a response.

7.2.2 Physical block size

The physical block size is defined as the number of 8-bit bytes that are capable of being written to, read from, or locked on the particular RF tag. For the purposes of this International Standard, the value is constrained to between 1 and 256 bytes.

NOTE: The block size relates to memory units on the RF tag and not necessarily to any constraint, such as a frame, on the air interface.

The physical block size shall be identified by any reasonable means using the Tag Driver and air interface features and is communicated to the Data Protocol Processor. The physical block size is not communicated across the application interface.

7.2.3 Number of blocks

The number of blocks is defined as the number of physical blocks in the user memory of the particular RF tag.

The number of blocks shall be identified by any reasonable means using the Tag Driver and air interface features and is communicated to the Data Protocol Processor. The number of blocks is not communicated across the application interface.

7.2.4 Application Family Identifier

The Application Family Identifier (**applicationFamilyId**) shall be defined by the application and be a single byte value with the following structure:

bits 8 to 5 shall define the **applicationFamily**
bits 4 to 1 shall define the **applicationSubFamily**

The full list of Application Family Identifiers are defined in ISO/IEC 15961.

The Application Family Identifier shall be used as a low level selection device to separate RF tags with the particular application features or classes of data content, based on the code values specified in ISO/IEC15961. This feature is co-managed with JTC1 SC17, so currently 60 of the byte values are available for RFID for item management.

7.2.5 Storage Format

The storage Format (**storageFormat**) shall be defined by the application and be a single byte value with the following structure:

bits 8 and 7	shall determine the **accessMethod** (see 7.1.2.4 of ISO/IEC 15961)
bit 6	is reserved
bits 5 to 1	shall determine the **dataFormat** (see 7.1.2.5 of ISO/IEC 15961)

Once an **accessMethod** has been specified for an RF tag, all subsequent additional data shall be stored using the same **accessMethod**, whether the **storageFormat** is locked or not locked.

NOTE: An **accessMethod** may be changed from a noDirectory structure to directory structure (see 7.3.2).

Once a **dataFormat** has been specified for an RF tag, all subsequent additional data shall be stored in one of two ways:

- If the new data has the same root-OID implied by the **dataFormat,** or if this is explicitly stored on the RF tag as a root-OID, then only the RELATIVE-OID needs to be encoded.

- Otherwise, the full OBJECT IDENTIFIER shall be encoded.

7.3 Configuring the Logical Memory

The size of the Logical Memory shall be defined using the physical block size and number of blocks from the system information (see 7.2). This can be perceived as a matrix with the physical block size determining the X-axis and the number of blocks determining the Y-axis. This matrix concept is used in the remainder of this International Standard. This configuration process creates an empty Logical Memory.

The content of the Logical Memory is structured differently from the transfer syntax exchanged with the application. This is to improve the encoding efficiency and the structuring rules in the Logical Memory by making use of bit encoding rules and specifying a standard sequence of the encoded elements.

The **accessMethod**, part of the system information, also determines the structure and processing through the Logical Memory (see sub-clauses below).

The Logical Memory consists of a structured sequence of bytes that represents a repeating sequence of:

- Precursor - This is typically 1 byte long and provides a bit-based meta-data structure of the encoded **objectId** and **object**.
- The length of the **objectId.**
- **ObjectId** - This is a formatted **objectId** that makes use of the RELATIVE-OID and **dataFormat** to reduce the encoding, but still require only a simple process to reconstruct the complete OBJECT IDENTIFIER.
- The length of the **object**
- **Object** - This is the compacted **object**.

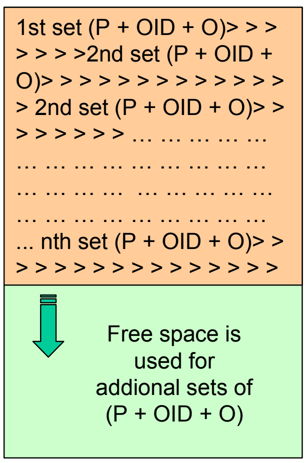
This is a general description and there are some variations to this defined in subsequent clauses. Clause 8.2 defines how the **object** shall be encoded. Clause 8.3 defines the formatting rules that shall be used to minimise the encoding of the **objectId** and to determine how the precursor is encoded.

The Data Protocol Processor needs to support a number of Logical Memories to enable transactions to take place with the required number of RF tags. The Logical Memory of each RF tag is referenced by its tag identifier.

Transactions across the air interface populate the Logical Memory only to the extent that is required to complete the transaction.

7.3.1 Non- directory structure of the Logical Memory

The **accessMethod = noDirectory** shall determine that the Logical Memory has the simplest structure. This consists of a repeating cycle of Precursor, formatted **ObjectId** and compacted data **object**. Any unused bytes on the Logical Memory are at the end. Figure 3 illustrates this.



Legend

P = Precursor

OID = ObjectId

O = (data) object

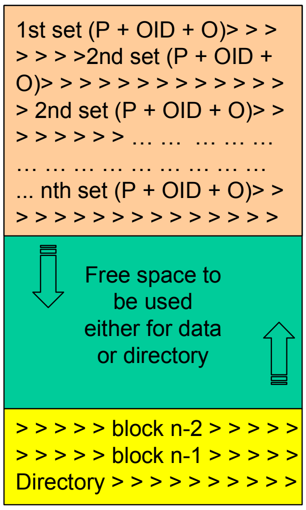
Figure 3 — Logical Memory Schematic - Non-directory Structure

7.3.2 Directory structure of the Logical Memory

The **accessMethod = directory** shall determine that the Logical Memory has a two part structure:

- The lower blocks are identical to the non-directory structure.
- The higher blocks contain the directory.

Directory entries shall be stored in the lowest byte address to the highest byte address sequence within a block, but in reverse block sequence starting from the last block. Figure 4 illustrates this.



Legend

P = Precursor

OID = ObjectId

O = (data) object

Figure 4 — Logical Memory Schematic - Directory Structure

Structuring the directory in this reverse sequence of blocks leaves unused blocks available either to encode additional data or directory until there are too few blocks remaining to encode new data. This flexibility allows for a small directory (i.e. few **objectIds**) with large data **objects**; or a large directory (i.e. many **objectIds**), each of which could have a small data **object**; or any other combination.

It is possible to begin with a non-directory Logical Memory structure, and some time later convert to a directory structure. This first requires all of the content of the RF tag's application memory to be transferred to the Logical Memory of the Data Protocol Processor. Then the **accessMethod** needs to be changed to indicate that a directory structure is now invoked, and a directory created. Finally the system info needs to be updated on the RF tag, and the content of the directory transferred to the RF tag.

7.3.3 Self mapping RF tags

The **accessMethod = selfMappingTag** indicates that integral processing capability to organise the data mapping is embedded within the RF tag. As such, it shall perform similar functions to that of the Data Formatter (see 8.3) of the Data Protocol Processor.

Data compaction shall be performed on the data **object** if required by a write command argument, and de-compaction shall be performed if the encodation for the **object** on the RF tag signals that it is compacted.

Processing by the Data Formatter shall be by-passed for both read and write processes. As such, the self mapping tag shall process and respond directly to application commands with the exception of data compaction.

Self mapping tags may use other processes than a precursor and directory to structure their encoded **objectIds** and **objects**.

NOTE: At the date of first publication of this International Standard, none of the artefacts demonstrated for ISO/IEC 18000 support such a sophisticated level of processing.

8 Data flows and processes

This clause specifies how the data, received and decoded from the transfer syntax from commands defined in ISO/IEC 15961, is processed for efficient encoding onto the Logical Memory using the Encoder sub-components of a Data Compactor and Formatter as described in 4.3.

Although the process can only be fully invoked through the Command/Response Unit, this clause focuses exclusively on the processing of the **objectId** and the (data) **object**.

8.1 Application data

The application shall provide data to the Data Protocol Processor in the form of an **objectId** and an associated (data) **object.** It shall also provide information, using the command argument **objectLock**, whether to lock the bytes in the Logical Memory (and in turn on the RF tag) associated with the **object** and **objectId.** The format of the application data shall be compliant with Clause 7.3 of ISO/IEC 15961.

NOTE: For the remainder of the International Standard, it will be assumed that the transfer mechanism from the application is fully compliant with the transfer encoding rules used in ISO/IEC 15961. Implementations that do not use all the specifications of ISO/IEC 15961 need to ensure that transfers are achieved in a completely reliable manner.

8.1.1 Processing data transferred from ISO/IEC 15961

The initial objective is to create a structured list (probably in tabular form as illustrated in Table 2) of the **objectId(s)** and **object(s)** being processed. The following sub-clauses specify the rules for encoding.

Table 2 — Tabular Format for Processing TransferEncoding of ISO/IEC 15961

ObjectId			Object		
Class Tag	**Length**	**ObjectId**	**Class Tag**	**Length**	**Object**

8.1.1.1 Processing the objectId

ISO/IEC 15961 (Clause 6.1.7) specifies that the OBJECT IDENTIFIER Type is used in the transfer encoding of the **objectId** from the application to the Data Protocol Processor. This has the Class Tag 6 (06_{HEX}).

As the OBJECT IDENTIFIER Type always has this value (06_{HEX}), the next byte value is the beginning of the length of the **objectId**.

The initial processing of the **objectId** is simply to transfer the number of bytes, defined by the preceding length component, to the formatted structure. Subsequent processing is required by the Data Formatter (see 8.3).

8.1.1.2 Processing the object

The specification of **object** in ISO/IEC 15961 reverts to the Universal Type OCTET STRING, so this is the only data format used in the transfer encoding from the application to the Data Protocol Processor. This has the Class Tag 4 (04_{HEX}).

As the OCTET STRING Type always has this value (04_{HEX}), the next byte value is the beginning of the length of the **object**.

NOTE: So that the OCTET STRING can be given different interpretations by the application, the command modules include various qualifiers that impact on the encoding on the Logical Memory.

The initial processing of the **object** is simply to transfer the number of bytes, defined by the preceding length component, to the formatted structure. Subsequent processing is required by the Data Compactor (see 8.2).

8.1.1.3 Length decoding

The length of the **objectId** and **object** shall be decoded from the transfer syntax using the rules defined in 6.3.

8.1.1.4 Processing the objectLock argument

The value of the command argument **objectLock** determines whether the encoded bytes representing the **object** and **objectId** are to be locked (and therefore intended to be permanently encoded) or not. The processing of this argument is addressed in 9.2.7.

8.1.2 Processing example

The following example illustrates the process defined in the sub-clauses above.

EXAMPLE

The complete encoding is: 06 05 28 FC 59 0A 30 04 09 41 42 43 31 32 33 34 35 36

1st byte 06_{HEX} = OBJECT IDENTIFIER Type

The remaining encoding is: 05 28 FC 59 0A 30 04 09 41 42 43 31 32 33 34 35 36

The next byte(s) define the length. The value 05_{HEX} is in the range 01_{HEX} to $7F_{HEX}$, so this single byte defines the length of the **objectId**. The length is 5 bytes.

The remaining encoding is: 28 FC 59 0A 30 04 09 41 42 43 31 32 33 34 35 36

The next five bytes are the OBJECT IDENTIFIER = 28 FC 59 0A 30

The remaining encoding is: 04 09 41 42 43 31 32 33 34 35 36

The next single byte 04_{HEX} is the Object Type = OCTET STRING Type

The remaining encoding is: 09 41 42 43 31 32 33 34 35 36

The next byte(s) define the length. The value 09_{HEX} is in the range 01_{HEX} to $7F_{HEX}$, so this single byte defines the length of the **object**. The length is 9 bytes.

The remaining 9 bytes of the encoding are the **object**: 41 42 43 31 32 33 34 35 36

The byte string can be placed in the appropriate format, e.g. as in a tabular format similar to that defined in 8.1.1, as illustrated in Table 3.

NOTE: An additional **objectId** and **object** pair have been added to this table to enable this more detailed example to be referred to for illustrating subsequent processes.

Table 3 — Process Example in Tabular Form

ObjectId			Object		
Class Tag	**Length**	**ObjectId**	**Class Tag**	**Length**	**Object**
06	05	28 FC 59 0A 30	04	09	41 42 43 31 32 33 34 35 36
06	05	28 FC 59 0A 0E	04	02	35 30

8.2 Data object processing

Data compaction is applied as a transformation to the data objects to reduce the number of bits which are transferred across the air interface and which are required to encode data in memory. The compaction shall be done according to the **compactParameter** (defined in 8.2.1) received in the ISO/IEC 15961 commands. The interpretation of the source data objects (e.g. conforming to particular character sets) can be ignored by the interrogator and while encoded on the RF tag. This is because the decode process is the inverse of the encode process, and the compacted data objects are restructured to their original forms when the tag is read.

The Data Compactor performs all the processes necessary to compact a data object and to determine the Compaction Type. The **objectId** remains unchanged and is not subject to any form of compaction. to enable it to be directly identifiable by the application, the transfer encoding, and the Logical Memory. The command argument **objectLock** remains unchanged and is passed through to the next stage of processing in the Data Formatter.

The length of the compacted **object** may be less than on input to the process. In addition, the way that length is stored on input from the application interface (i.e. subject to the transfer encoding rules defined in ISO/IEC 15961 and decoded by the process defined in 6.3 of this International Standard), is different from the way that it is encoded in the Logical Memory (see 8.2.4). This is to reduce the number of bytes required.

Figure 5 illustrates the data flow.

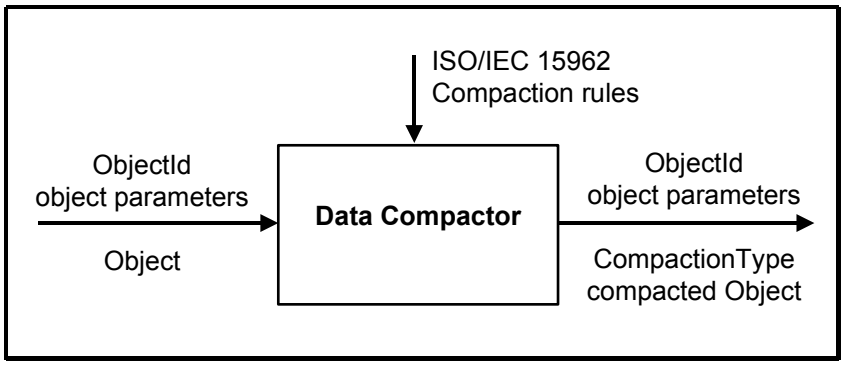


Figure 5 — Data Flow Model: Data Compactor

8.2.1 Compaction process

The command argument **compactParameter** determines whether the **object** is subject to the compaction process or not, based on the following integer values.

0 **applicationDefined:** The data object is read by and passed through the Data Compactor without processing, but is assigned the Compaction Type Code 000_{BIN}.

1. **compact:** The data object is read by and passed through the Data Compactor to be compacted as efficiently as possible and assigned the appropriate Compaction Type Code in the range 001_{BIN} to 110_{BIN}.

2. **utf8Data:** The data object is read by and passed through the Data Compactor without processing, but is assigned the Compaction Type Code 111_{BIN} to indicate that it is compliant with the UTF-8 transformation of ISO/IEC 10646.

3 to 15 reserved

The **object** is read by and passed through the Data Compactor.

1. The data object itself is transformed to its compacted form. If the command argument **compactParameter:**

   a. is set to 1 (**compact**) the data object input string is compacted (see 8.2.2 and Annex C for the detailed processes).

   b. is set to 0 (**applicationDefined**), or to 2 (**utf8Data**) the input string is not compacted but the data object is still processed through step 2 and step 3.

2. The 3-bit Compaction Type Code is assigned (see 8.2.3).

3. The length of the compacted data **object** is defined.

8.2.2 Compaction Schemes

Data compaction shall be applied to each entire data object. When compaction is requested by the **compactParameter**, the selection of the particular data compaction scheme is determined by parsing the bytes in the data object and analysing their values as defined in Table 4, which shows the Compaction Schemes in sequence of preferred application, starting with the most efficient. The compaction rules for all the compaction Schemes are defined in Annex C.

Table 4 — Determining the Data Compaction Scheme

All bytes in the range (HEX)	Secondary Conditions	Use Compaction Scheme	Refer to Annex
30 to 39	1. Leading byte(s) $\neq 30_{HEX}$ 2. Length of object > 1 byte 3. Length of object ≤ 19 bytes	integer	C.1
30 to 39	Length of object > 1 byte	numeric	C.2
41 to 5F	Length of object > 2 bytes	5 bit code	C.3
20 to 5F	Length of object > 3 bytes	6 bit code	C.4
00 to 7E	Length of object > 7 bytes	7 bit code	C.5
00 to FF	N/A	octet string	C.6

If the original character string is too short to satisfy the secondary condition for length, the octet string scheme shall be used. If the other secondary conditions for integer compaction cannot be met, then numeric compaction shall be used.

For illustration, the compaction types refer to character representations of ISO/IEC 646 (see Annex D) for values 00 to 7E. However, the compaction schemes apply to the string of byte values, irrespective of their interpretation. The precise interpretation is defined by the definition of the **objectId** given externally in the application.

8.2.3 Compaction Type codes

The codes for the **Compaction Type** are defined in Table 5. One of these is applied to the data object after it has been processed by the Data Compactor. On subsequent reading from the RF tag, this is used to identify how the data object shall be de-compacted.

Table 5 — Compaction Type Codes

Code Value		Name	Description
Decimal	Binary		
0	000	application defined	as presented by the application
1	001	integer	Integer
2	010	numeric	Numeric String (from "0"..."9")
3	011	5 bit code	Uppercase alphabetic
4	100	6 bit code	Uppercase, numeric, etc.
5	101	7 bit code	US ASCII
6	110	octet string	unaltered 8-bit
7	111	UTF-8 string	External compaction of ISO/IEC 10646

8.2.4 Encoding the length of the compacted object

The length of all **objects** on output from the compaction process (including the **objects** not intended for compaction, or not achieving a compacted state) shall be determined as follows:

1. If the length is between 0 and 127 bytes, the length is encoded in one byte with the lead bit = 0

   0bbbbbbb
   where bbbbbbb = length in bytes

2. If the length is between 128 and 16383 bytes, the length is encoded in two bytes as follows:

   a. Set the first bit of the lead byte = 1 and the first bit of the second byte = 0.
      1bbbbbbb 0bbbbbbb
   b. Convert the length (in bytes) to its binary value.
   c. Encode the value in the bits 7 to 1 of each byte of the length encoding.

3. If the length is between 16384 and 2097151, the length is encoded in three bytes as follows:

   a. Set the first bit of the lead byte = 1 and the first bit of other bytes = 0.
      1bbbbbbb 0bbbbbbb 0bbbbbbb
   b. Convert the length (in bytes) to its binary value.
   c. Encode the value in the bits 7 to 1 of each byte of the length encoding.

NOTE: Although probably an unrealistic requirement the rule can be extended to cover any length.

8.2.5 Example of encoding for the Logical Memory after compaction

Annex E.3 illustrates an example of the encoding after the compaction process is applied

8.3 Data formatting

Data formatting shall be applied as a mapping process to the compacted data **object** and its **objectId** to place the bytes on the Logical Memory in an efficient manner to access data and reduce memory space required.

The Data Formatter performs all the processes necessary to convert the input from the Data Compactor into the bytes stored on the Logical Memory. This includes formatting to the **accessMethod** (which includes the use of a directory and non-directory structure) and to the **dataFormat** (which enables relative object identifiers to be used to save on the use of memory). It also includes formatting to satisfy the requirements of the **objectLock** argument.

The Data Formatter and Logical Memory are both integral modules of the Data Protocol Processor. As such, they shall not be de-coupled from other modules of the Data Protocol Processor. They are presented as separate modules in this clause so that the functions that need to be performed can be understood. The modular structure could also help in identifying problems of non-conformance. Figure 6 illustrates the data and information flow.

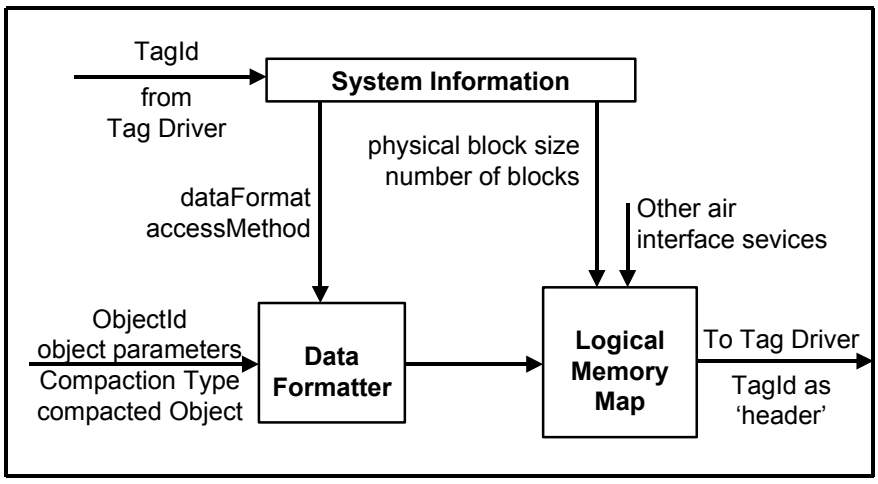


Figure 6 — Data Flow Model : Data Formatter Module

8.3.1 Data Formatter Functions

The Data Formatter receives from the Data Compactor a set of:

- **objectIds** as received from the application services
- compacted data **objects**
- other parameters, including the argument **objectLock** that requires processing by the Data Formatter, and others that affect the transaction across the air interface

These are received in a format as illustrated in Table E.3 of Annex E.

A prime function of the Data Formatter is to identify whether a root-OID can be used to carry out defined processes to reduce the encoding space required for each **objectId**. This process is controlled by the **dataFormat** as defined by the system information.

Another function is to map the bytes in the Logical Memory to support either a directory or non-directory structure. The process is controlled by the **accessMethod** as defined by the system information.

NOTE: The **accessMethod = selfMappingTag** is not yet supported by ISO/IEC 15961 and this International Standard, other than to by-pass the processes of the Data Formatter. If such an RF tag is introduced that requires processing through the Data Formatter, then this will be addressed in future revisions of this International Standard.

8.3.2 Formatting the objectId

The Data Formatter shall take the values of the **dataFormat** element from system information (as defined in 7.2.5) and process the set of **objectId** values as specified below. The processes described below are based on a typical write command being processed on an RF tag. Where there are significant variants these are described. The processing of the Precursor is addressed in 8.3.3 and 8.3.4.

A. If **dataFormat = notFormatted (0)**

An error has occurred and formatting is not possible.

B. If **dataFormat = fullFeatured (1)**

Each **objectId** shall be encoded as a full OBJECT IDENTIFIER.

C. If **dataFormat = rootOidEncoded (2)**

1. Parse the set of **objectId** values to identify any common root-OID that shall have at least two arcs. If the RF tag already has an encoded root-OID, establish whether this can be applied to the new **objectIds**. Once a common root-OID has been identified, any other **objectId** value with a different root shall be encoded as a full OBJECT IDENTIFIER.

2. Create the common root-OID, if one is not already encoded on the RF tag.

3. Strip out the common root-OID from each **objectId** to create a RELATIVE-OID from the remaining arcs.

4. If Steps 2 and 3 are not possible, an error has occurred and formatting is not possible**.**

5. Each other **objectId** that cannot have the root-OID stripped remains unchanged and is encoded as an OBJECT IDENTIFIER.

An example of the input as shown in Annex E.3, subsequently processed with a **data format = rootOidEncoded** is illustrated in Annex E.4.1

D. If **dataFormat = iso15434 (3), or**
iso6523 (4), or
iso15459 (5), or
iso15961combined (8), or
ean-ucc (9), or
di (10), or
upu (11), or
iata (12)

1. Parse the set of **objectId** values to identify the common root-OID, as defined below:

dataFormat	common root-OID	
iso15434 (3)	{1 0 15434}	{28 F8 4A $_{HEX}$}
iso6523 (4)	{1 0 6523}	{28 B2 7B $_{HEX}$}
Iso15459 (5)	{1 0 15459}	{28 F8 63 $_{HEX}$}
iso15961combined (8)	{1 0 15961}	{28 FC 59 $_{HEX}$}
ean-ucc (9)	{1 0 15961 9}	{28 FC 59 09 $_{HEX}$}
di (10)	{1 0 15961 10}	{28 FC 59 0A $_{HEX}$}
upu (11)	{1 0 15961 11}	{28 FC 59 0B $_{HEX}$}
iata (12)	{1 0 15961 12}	{28 FC 59 0C $_{HEX}$}

2. If Step 1 is not possible an error has occurred and formatting is not possible.

3. Strip out the common root-OID from Step 1 from each **objectId** to create a RELATIVE-OID from the remaining arcs.

   NOTE: The root-OID is not encoded, but implied by the **dataFormat**

4. Each other **objectId** that cannot have the root-OID stripped remains unchanged and is encoded as an OBJECT IDENTIFIER.

An example of the input as shown in Annex E.3, subsequently processed with a **data format** = **di** is illustrated in Annex E.4.2

8.3.3 The Precursor for dataFormat not equal 2

The Precursor shall be one byte that precedes the formatted **objectId** and compacted **object**. In some circumstances it may be longer.

The structure of the Precursor is bit based from bit 8 to bit 1, except for Precursor value 00_{HEX}.

- Bit 8: the offset and expansion bit
  if bit 8 = 0 no offset is present
  if bit 8 = 1 an additional byte follows as part of the Precursor
- Bits 7 to 5: the Compaction Type Code (see 8.2.3)
- Bits 4 to 1: used to directly encode some RELATIVE-OID values
  - ► values 0001 to 1110_{BIN} shall encode a RELATIVE-OID that has only one arc and that arc has a value 1 to 14
  - ► value 1111 indicates that the RELATIVE-OID, OBJECT IDENTIFIER or root-OID is directly encoded in subsequent bytes

If the Precursor has the value 00_{HEX}, then it shall act as a terminator of the **object** list and of the directory on the Logical Memory.

8.3.4 The Precursor for the root-OID for dataFormat = 2

The **dataFormat = rootOidEncoded (2)** requires the root-OID to be directly encoded. The Precursor for the root-OID only shall be 1 byte that precedes the root-OID; there is no associated compacted **object**. This particular Precursor structure applies to the root-OID being encoded in the directory and in the data set area. In some circumstances the Precursor may be longer.

The structure of the Precursor is bit based from bit 8 to bit 1:

- Bit 8: the offset and expansion bit
  if bit 8 = 0 no offset is present
  if bit 8 = 1 an additional byte follows as part of the Precursor
- Bits 7 to 1: the length of the root-OID (in bytes) between 1 to 126
  bbbbbbb = length of the root-OID

The Precursor for all other data sets in Logical Memory structures with **dataFormat =2** shall be as that defined in 8.3.3.

8.3.5 Encoding the RELATIVE-OID

The rules for encoding a RELATIVE-OID depend on the number of arcs and the value of the final arc, as defined below.

8.3.5.1 Single arc RELATIVE-OID value 1 to 14

The final byte of the **objectId** (representing the single arc value) transferred from the application interface shall have a value 01 to $0E_{HEX}$. The single arc value shall be encoded in bits 4 to 1 of the Precursor as defined in 8.3.3. No additional bytes shall be encoded in the Logical Memory for this **objectId**.

8.3.5.2 Single arc RELATIVE-OID value 15 to 127

The final arc of the **objectId** (representing the single arc value) transferred from the application interface shall have a value 0F to $7F_{HEX}$. The single arc shall be encoded in a single byte as follows:

$0bbbbbbb_{BIN}$

Where bbbbbbb = arc value -15

EXAMPLE:

RELATIVE-OID = 23
bbbbbbb = 23 - 15
= 00001000_{BIN}
= 08_{HEX}

8.3.5.3 Other RELATIVE-OID values

Irrespective of the number of arcs and the value of those arcs, other RELATIVE-OID values shall be processed as follows:

1. Record the byte string that represents the RELATIVE-OID value as derived from the process of formatting the **objectId** (see 8.3.2).

2. Count the number of bytes in the RELATIVE-OID.

3. If the length of the RELATIVE-OID is between 1 to 16 bytes, continue, else go to step 4.
   a. Assign 1 byte for length encoding.
   b. Assign value 100 for bits 8 to 6 to act as an indicator.
   c. Encode the length (in bytes) of the RELATIVE-OID in the remaining 5 bits:
      bbbbb-1 = length of RELATIVE-OID
   d. Encode the RELATIVE-OID in the subsequent bytes.

   NOTE: This procedure allows a RELATIVE-OID with a single arc value 0 (presented as 00_{HEX}) to be encoded.

4. If the length of the RELATIVE-OID is between 17 to 126 bytes:
   a. Assign 2 bytes for length encoding.
   b. Assign value 101 for bits 8 to 6 of the first byte to act as an indicator.
   c. Encode the length (in bytes) of the RELATIVE-OID in bits 7 to 1 of the second byte.
      10100000 0bbbbbbb
   d. Encode the RELATIVE-OID in the subsequent bytes.

   EXAMPLE:

   RELATIVE-OID = {1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 }

   From application
   (29 bytes long) = 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09_{HEX}

   Encoding = A0 1D 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09_{HEX}

The first two bytes define the length, but all the others are as provided by the application

8.3.6 Encoding the OBJECT IDENTIFIER

Irrespective of the number of arcs and the value of those arcs, the OBJECT IDENTIFIER values shall be processed as follows:

1. Record the byte string that represents the OBJECT IDENTIFIER value as derived from the process of formatting the **objectId** (see 8.3.2).

2. Count the number of bytes in the OBJECT IDENTIFIER.

3. If the length of the OBJECT IDENTIFIER is between 1 to 32 bytes, continue, else go to step 4.
   a. Assign 1 byte for length encoding.
   b. Assign value 110 for bits 8 to 6 to act as an indicator.
   c. Encode the length (in bytes) of the OBJECT IDENTIFIER in the remaining 5 bits.
      bbbbb -1 = length of OBJECT IDENTIFIER
   d. Encode the OBJECT IDENTIFIER in the subsequent bytes.

      EXAMPLE:

      OBJECT IDENTIFIER = {0 1 15961 9 1}

      From application
      (5 bytes) = 28 FC 59 09 01_{HEX}

      Encoding = C4 28 FC 59 09 01_{HEX}

4. If the length of the OBJECT IDENTIFIER is between 33 to 127 bytes:
   a. Assign 2 bytes for length encoding.
   b. Assign value 111 for bits 8 to 6 of the first byte to act as an indicator.
   c. Encode the length (in bytes) of the OBJECT IDENTIFIER in bits 7 to 1 of the second byte.
      11100000 0bbbbbbb
   d. Encode the OBJECT IDENTIFIER in the subsequent bytes.

      EXAMPLE:

      OBJECT IDENTIFIER = {0 1 15961 9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9}

      From application
      (33 bytes) = 28 FC 59 09 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09_{HEX}

      Encoding = E0 21 FC 59 09 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09_{HEX}

      The first two bytes define the length, but all the others are as provided by the application

8.3.7 Encoding the root-OID for dataFormat = rootOidEncoded (2)

A root-OID up to 126 bytes long can be encoded. The value of the length is directly encoded in the Precursor (see 8.3.4), so the root-OID immediately follows the Precursor. The value of the root-OID is exactly as the byte string derived from process C defined in 8.3.2. It shall be encoded at the start of the Logical Memory.

As the root-OID has no associated data **object**, the length of the data shall be set to zero.

8.3.8 Encoding the object and its length

The length and value of the (compacted) **object** are given by the output of the compaction process (see 8.2.4).

The length is encoded immediately following the formatted **objectId**. The compacted **object** is encoded next in sequence.

8.3.9 The offset byte

The offset byte shall be required if the block(s) containing the Precursor, **objectId, object** and their associated length encoding needs to be block aligned. This could be necessary because this data set or another requires to be locked.

The offset byte shall encode the value of the number of pad bytes required to follow the end of the **object** so as to place the next Precursor or terminator at the beginning of the next block.

NOTE: As the maximum block size is 256 bytes (see 7.2.2), and the Precursor plus offset require two bytes, the maximum value for the offset is FE_{HEX}.

The offset value 00_{HEX} is valid. It is used when the offset itself provides the padding feature, i.e. when the data set without the offset is one byte short of reaching a block boundary.

The offset value FF_{HEX} shall be used to indicate that the offset is followed by a Precursor expansion byte (see 8.3.10).

8.3.10 The Precursor expansion byte

If the offset byte has the value FF_{HEX}, it shall be followed by the Precursor expansion byte. At present, this only has a limited specification. Bit 7 to 1 are reserved for future expansion of the Precursor. Until this International Standard is amended, the Precursor expansion byte shall not be used.

When it is more fully specified, if bit 8 = 0, there is no offset byte to follow. If bit 8 = 1, then an additional offset byte shall follow and values 00 to FE_{HEX} shall be used to indicate the value of the offset.

NOTE: This sub-clause is included to show that the Precursor feature has scope for expansion if required in the future.

8.3.11 The directory structure

The encodation of the data sets shall be exactly as for the non-directory structure. The directory itself has the following sequence:

- If **dataFormat = rootOidEncoded (2)**:
  - ► Precursor of the root-OID,
  - ► The root-OID.
  - ► The length of the **object** encoded as $00_{HEX.}$
- A repeating cycle of data set identities, each consisting of:
  - ► Precursor for the encoded **objectId** and **object**.
  - ► The length of the **objectId.**
  - ► The address, as byte value(s), of the start of the Precursor associated with the Precursor of the data set.
- A terminator byte

The directory should not be locked, because if so done, it renders it impossible to update **objects** and **objectIds** on the Logical Memory. However, the data set of Precursor, **objectId** and **object** may or may not be locked to suit the needs of the application.

8.3.12 Addressing from the directory

The main purpose of the directory is to provide a mechanism to point to the specific location of a particular data set of Precursor, **objectId** and **object**. This is achieved by having a directory address that points to the first byte (i.e. the Precursor) of the individual data set.

The address shall be the ordinal byte position, with the first byte of the Logical Memory being byte 1. The directory address is encoded over one or more bytes: one if the address is 1 to 127, two for an address up to 16383, three for an address up to 2097151, and so on. The rules for defining the address are identical to the rules for the length of the **object** (see 8.2.4).

8.3.13 Structures of Logical Memory

All the preceding sub clauses of 8.3 provide rules that result in different structures of Logical Memory. Annex F provides schematic illustrations of the different structures. These could be useful in preparing overall decode rules.

8.4 Decoding the Logical Memory

The following sub clauses provide advice and rules to decode particular components of the Logical Memory. The simpler decode functions that can easily be determined from the encoding rules are not covered.

8.4.1 Overall decode strategy

The **storageFormat** is the prime key to decoding. Particular values determine whether a directory is present and this enables a three-stage search to be undertaken.

1. to read the entire directory.
2. to parse the directory to locate the address of the required data set.
3. to read the data set starting from that address.

In contrast, a non-directory structure requires continual processing of the data set area until the desired **objectId** is found.

The **storageFormat** also provides information about the root-OID, particularly whether this is implicitly or explicitly encoded.

The parsing process is continually looking forward, whether this is for reading the directory, the data sets in a non-directory structure, or a particular data set starting with the Precursor and ending with the **object**.

Each data set is decoded in the sequence:

Precursor
objectId
length of **object**
object

with the appropriate variants being correctly processed. If the **object** is not required, the value for length of **object** can be used to skip forward to the next Precursor. A similar, but slightly more complex, process is possible to invoke to skip over the **objectId**. In this case, the first byte(s) of the encoded **objectId** need to be parsed to determine the length and type of **objectId**. These points are more fully described below.

8.4.2 Decoding the storageFormat

The following **storageFormat** values are in the format m, n where m = **accessMethod** (0-3) and n = **dataFormat** (0-31). This advice only applies to those values already assigned in this International Standard and in ISO/IEC 15961.

The following values indicate particular structures and processes:

0,1 -- non-directory, with each **objectId** presented as a full OBJECT IDENTIFIER

0,2 -- non-directory with an explicitly encoded root-OID in the first logical position. All subsequent RELATIVE-OID values are prefixed by the root-OID. Full OBJECT IDENTIFIERS may appear in the encodation.

0,n (n = 3,4,5,8,9,10,11,12) -- non-directory with the root-OID defined by the **dataFormat.** All subsequent RELATIVE-OID values are prefixed by the root-OID. Full OBJECT IDENTIFIERS may appear in the encodation.

1,1 -- directory with each **objectId** presented as a full OBJECT IDENTIFIER in both the directory and the data set area.

1,2 -- directory with an explicitly encoded root-OID in the first logical position of the directory and also in the data set area. All subsequent RELATIVE-OID values are prefixed by the root-OID. Full OBJECT IDENTIFIERS may appear in the encodation.

1,n (n = 3,4,5,8,9,10,11,12) -- directory with the root-OID defined by the **dataFormat**. All subsequent RELATIVE-OID values are prefixed by the root-OID. Full OBJECT IDENTIFIERS may appear in the encodation.

Each of these values can also be used to identify a Logical Memory structure as given in Annex F.

NOTE: As **accessMethods** and **dataFormats** are assigned, additional rules could be applied, or the rules defined above be applied to additional assignments.

8.4.3 Decoding the Precursor

The following bit-based structure applies to the Precursor, with two exceptions defined in the following sub clauses.

bits

8 If = 0 There is no expansion nor offset byte to follow.

If = 1 There is an expansion byte (see 8.3.10) which may be followed by a second offset byte.

7 to 5 Defines the Compaction Type Code (see 8.2.3). Decompaction is only required if the command calls for the particular **object** to be returned.

4 to 1 Values 0001 to 1110 represent the value of the single RELATIVE-OID arc value 1 to 14. In this case, the Precursor is followed by the byte(s) defining the length of the **object.** The **objectId** is a Null field.

Value 1111 indicates that the Precursor is followed by an encoded **objectId** (of any permitted type).

8.4.3.1 The Terminator byte

If the Precursor has the value 00_{HEX} it acts as a Terminator to signal the end of the directory and/or the end of the encoding of the data sets.

8.4.3.2 The Precursor for the explicitly encoded root-OID

If the **storageFormat** is 0,2 the Precursor in the first logical byte position shall have the following structure:

bits

8 If = 0 There is no expansion nor offset byte to follow.

If = 1 There is an expansion byte (see 8.3.10) which may be followed by a second offset byte.

7 to 1 Defines the length (in bytes) of the root-OID.

If the **storageFormat** is 1, 2 the Precursor in the first logical byte position of the directory and of the data set area shall have the structure as above.

All other Precursors in **storageFormat**, 0,2 and 1,2 shall comply with the decode process defined in 8.4.3.

8.4.4 Decoding the leading byte(s) of the encoded objectId

Generally, the encoding of the **objectId** immediately follows the Precursor. The only exceptions to this are when the Precursor itself encodes the **objectId** (i. e. as for a low value single arc RELATIVE-OID), and where an offset byte is inserted after the Precursor.

The first byte(s) of the encoded **objectId** identify the type of **objectId** and the length, and sometimes the value of this, as defined below:

First byte value (HEX)

00 to 70 This encodes the value of a RELATIVE-OID that has a single arc (see 8.3.5.2). To calculate the value of the arc for the application interface add $0F_{HEX}$ to that in the Logical Memory.

This byte is followed by the byte(s) encoding the length of the encoded **object**.

80 to 9F This encodes the length of a RELATIVE-OID (see 8.3.5.3) for lengths between 1 to 16 bytes. Bits 5 to 1 encode this, and the length = bbbbb - 1.

This byte is followed by the RELATIVE-OID.

A0 This encodes the length of a RELATIVE-OID using this lead byte and the next byte (see 8.3.5.3), for lengths between 17 to 126 bytes. Bits 7 to 1 of the second byte directly encode the length.

This pair of bytes is followed by the RELATIVE-OID.

C0 to DF This encodes the length of an OBJECT IDENTIFIER (see 8.3.6) for lengths between 1 to 32 bytes. Bits 5 to 1 encode this; and the length = bbbbb - 1

This byte is followed by the OBJECT IDENTIFIER.

E0 This encodes the length of an OBJECT IDENTIFIER using this lead byte and the next byte (see 8.3.6) for lengths between 33 to 127 bytes. Bits 7 to 1 of the second byte directly encode the length.

This pair of bytes is followed by the OBJECT IDENTIFIER.

9 The Command / Response unit

The Command/Response unit is responsible for processing all application commands and providing all responses to the application system from the RF tag(s) with which it is communicating. This unit can be considered conceptually as some form of executive software that receives application commands as the transfer syntax defined in ISO/IEC 15961. It then calls for the data compaction encode and decode functions and the data formatting encode and decode functions to process the **objects** and **objectIds**. The command arguments concerned with processing (e.g. lock functions, conditional selections and so on) call for interfaces with the Tag Driver and, in turn, the particular ISO/IEC 18000 RF tag command sets. Throughout this process, the Command/Response unit is continually monitoring and updating the Logical Memory of each RF tag with which it is maintaining a communication link.

As the data flow and processing functions have been fully defined in Clause 8, the following sub clauses will focus - on a command-by-command basis - on the additional command arguments and response arguments that have not been addressed previously in this International Standard. In this sense, this clause is closely related to Clause 8 of ISO/IEC 15961. Reference to that standard is essential to developers of the Command/Response unit. This is because the Abstract Syntax describes the inter-relationship of the various command arguments, parameters and responses. It should also be borne in mind that the actual transfers of application commands and responses shall be based on the transfer syntax specified in ISO/IEC 15961.

9.1 Commands

Commands and responses are presented as linked pairs of modules. These are identified using the OBJECT IDENTIFIER type as specified in Clause 8.1 of ISO/IEC 15961. The processing of individual commands is defined in the following sub-clauses.

9.1.1 Configure Application Family Identifier command

The Application Family Identifier is held on the RF tag as part of the system information, separate from the Logical Memory Map. Its initial configuration. and possible change, is called for by this application command. The command (see 8.5 of ISO/IEC 15961) assumes that the Application Family Identifier is being configured for the first time or, alternatively, is an instruction to overwrite the existing value (if possible).

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard:

afiLock (see 9.2.1)
completionCode (see 9.3)
executionCode (see 9.4)

9.1.2 Configure Storage Format command

The Storage Format is held on the RF tag as part of the system information, separate from the Logical Memory Map. Its initial configuration. and possible change, is called for by this application command. The command (see 8.6 of ISO/IEC 15961) assumes that the Storage Format is being configured for the first time or, alternatively, is an instruction to overwrite the existing value (if possible).

The following elementNames in the abstractsyntax require processing as defined elsewhere in this International Standard:

completionCode (see 9.3)
executionCode (see 9.4)
storageFormatLock (see 9.2.8)

9.1.3 Inventory Tags command

The command (see 8.7 of ISO/IEC 15961) seeks to identify the **tagIds** of a defined subset of the RF tags present in the operating field.

The first selection is achieved by the command specifying the **applicationFamilyId**, with the following values having particular effects:

xy_{HEX} (where x is non-zero, currently in the range 9..C; and where y is non-zero, currently in the range (1..F) Identifies RF tags belonging to a particular system (typically determined by a particular root-OID) and containing particular data.

00_{HEX} Identifies all RF tags in the operating field.

The second selection criterion is concerned with the **identifyMethod** and **numberOfTags** (see 9.2.4), which are required to be identified to comply with the basic requirements of the first criterion.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

completionCode (see 9.3)
executionCode (see 9.4)
identifyMethod (see 9.2.4)
numberOfTags (see 9.2.4)

9.1.4 Add Single Object command

The command (see 8.8 of ISO/IEC 15961) instructs the interrogator to add an **object, objectId** and associated Precursor to the Logical Memory Map of a particular RF tag.

The command argument **compactParameter** defines the processes through the Data Compactor (see 8.2). Other elementNames in the abstract syntax require processing as defined elsewhere in this International Standard:

avoidDuplicate (see 9.2.2)
completionCode (see 9.3)
executionCode (see 9.4)
objectLock (see 9.2.7)

9.1.5 Delete Object command

The command (see 8.9 of ISO/IEC 15961) instructs the interrogator to delete an **object, objectId** and associated Precursor from the Logical Memory Map of a particular RF tag.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

checkDuplicate (see 9.2.3)
completionCode (see 9.3)
executionCode (see 9.4)

9.1.6 Modify Object command

The command (see 8.10 of ISO/IEC 15961) instructs the interrogator to modify the value of an **object** in the Logical Memory Map of a particular RF tag. In turn, this is likely to result in updating the Precursor (encoding and position) and the position of the **objectId.**

If the **accessMethod = directory**, the directory first shall be read into the Logical Memory Map and checked to confirm that the specified **objectId** is present and its address identified. If the **accessMethod = noDirectory**, the data sets of Precursor, **objectId** and **object** shall be read into the Logical Memory Map until the specified **objectId** is found. With both types of **accessMethod**, the complete **object** shall be deleted. The search process shall continue until the Logical Memory Map has been processed to identify all other instances of data sets associated with the specified **objectId**. If any duplicates are found, the appropriate error shall be returned.

If only one instance of the **objectId** was found, there is now an empty byte string for the modified object to be inserted. Three options are possible:

1. If the modified **object** is the same length as the previous encodation, it can be written to the same sequence of bytes. The Precursor could still need to be changed to identify a different compaction scheme.

2. If the modified object is shorter than the previous encodation, it can be written to the same sequence of bytes. The Precursor shall need to be changed and an offset encoded.

3. If the modified object is longer, the entire Logical Memory Map shall be re-constructed.

If the modified **object** is to be locked where the previous encodation was not, the entire Logical Memory Map could need to be re-constructed.

The command argument **compactParameter** defines the processes through the Data Compactor (see 8.2). Other elementNames in the abstract syntax require processing as defined elsewhere in this International Standard:

completionCode (see 9.3)
executionCode (see 9.4)
objectLock (see 9.2.7)

9.1.7 Read Single Object command

The command (see 8.11 of ISO/IEC 15961) instructs the interrogator to return an **object** associated with a specified **objectId** from the Logical Memory Map of a particular RF tag. The response has the **object**, **compactParameter** and **lockStatus**. Returning the **compactParameter** requires any compacted object bytes to be de-compacted before being returned as **decompactedData (15)**. Also, any bytes encoded as **applicationDefined (0)** or **utf8Data (2)** are returned with that same status. Returning the **lockStatus** requires processing by the Tag Driver to identify from RF tag features that the relevant blocks are locked.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

checkDuplicate (see 9.2.3)
completionCode (see 9.3)
executionCode (see 9.4)
lockStatus (see 9.2.5)

9.1.8 Read ObjectIds command

The command (see 8.12 of ISO/IEC 15961) instructs the interrogator to read all **objectId** values from the Logical Memory Map of a particular RF tag. This command can be used as a first stage to another more specific command. It can be used for "housekeeping", for example, to identify duplicate **objectIds**. As such, duplicate **objectIds**, if found, do not signal an error.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

completionCode (see 9.3)
executionCode (see 9.4)

9.1.9 Read All Objects command

The command (see 8.13 of ISO/IEC 15961) instructs the interrogator to return all **objectIds** and **objects** from the Logical Memory Map of a particular RF tag. The response has the completely structured list of each instance (including duplicates) of **objectId, object, compactParameter** and **lockStatus**. Returning the **compactParameter** requires any compacted object bytes to be de-compacted before being returned as **decompactedData (15)**. Also, any bytes encoded as **applicationDefined (0)** or **utf8Data (2)** are returned with that same status. Returning the **lockStatus** requires processing by the Tag Driver to identify from RF tag features that the relevant blocks are locked.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

checkDuplicate (see 9.2.3)
completionCode (see 9.3)
executionCode (see 9.4)
lockStatus (see 9.2.5)

9.1.10 Read Logical Memory Map command

The command (see 8.14 of ISO/IEC 15961) instructs the interrogator to transfer the complete byte stream represented in the Logical Memory Map of a particular RF tag. The response shall bypass the Data Formatter and Data Compactor and return an uninterpreted byte stream, including an unstructured directory, if present.

The main purpose of this command is to enable subsequent diagnostic analysis to be undertaken.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

completionCode (see 9.3)
executionCode (see 9.4)

9.1.11 Inventory And Read Objects command

The command (see 8.15 of ISO/IEC 15961) utilises the basic search functions of the Inventory command (see 9.1.3). In addition, the command argument **objectIdList** is used to define a common list of **objectIds** to be read from each RF tag that satisfies the search criterion. If the **objectIdList** is NULL then all **objectIds** and **objects** are returned for each RF tag meeting the selection criterion.

The response has a completely structured list of **objectId, object, compactParameter** and **lockStatus**. Returning the **compactParameter** requires any compacted object byte to be de-compacted before being returned as **decompactedData (15)**. Also, any bytes encoded as **applicationDefined (0)** or **utf8Data (2)** are returned with the same status. Returning the **lockStatus** requires processing by the Tag Driver to identify from the RF tag features that the relevant blocks are locked.

The response is similar to that of the Read All Objects command (see 9.1 9) except that only a partial list of **objectIds** may be specified to be returned. No error has occurred if a requested **objectId** is not found; the response simply excludes that nominated **objectId**. Duplicate entries are also permitted. The main difference in the response from this Inventory And Read Objects command and that of 9.1.9, is that multiple **tagIds** (and hence multiple Logical Memories) shall need to be supported.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

completionCode (see 9.3)
executionCode (see 9.4)
identifyMethod (see 9.2.4)
numberOfTags (see 9.2.4)

9.1.12 Erase Memory command

The command (see 8.16 of ISO/IEC 15961) instructs the interrogator to re-set to zero the entire Logical Memory Map of a particular RF tag. This includes the directory if this is defined as the **accessMethod**. All blocks shall be re-set except those that are locked. In this case the **completionCode = blocksLocked (17)** shall be returned.

Subsequent processing by the application, possibly by reading all the locked blocks, can be invoked to determine whether the RF tag is still usable. For example, the blocks that are locked could contain data permanently assigned to the item and the unlocked blocks contain transitory data.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

completionCode (see 9.3)
executionCode (see 9.4)

9.1.13 Get Application-based System Information command

The command (see 8.17 of ISO/IEC 15961) instructs the interrogator to read the system information and return the values of the **applicationFamilyId** and of the **storageFormat** for a particular RF tag. The process bypasses the Logical Memory.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

completionCode (see 9.3)
executionCode (see 9.4)

9.1.14 Add Multiple Objects command

The command (see 8.18 of ISO/IEC 15961) instructs the interrogator to add a sequence of **object, objectId** and associated Precursor to the Logical Memory Map of a particular RF tag. The command arguments **avoidDuplicate, compactParameter** and **objectLock** are applied to individual **objectIds** and **objects** and not to the complete set. Therefore, the organisation in the Logical Memory Map can be different from sequence presented in the command.

The command argument **compactParameter** defines the processes through the Data Compactor (see 8.2). Other elementNames in the abstract syntax require processing as defined elsewhere in this International Standard:

avoidDuplicate (see 9.2.2)
completionCode (see 9.3)
executionCode (see 9.4)
objectLock (see 9.2.7)

9.1.15 Read Multiple Objects command

The command (see 8.19 of ISO/IEC 15961) instructs the interrogator to return a sequence of **object** associated with a specified **objectId** from the Logical Memory Map of a particular RF tag. The command argument **checkDuplicate** is applied to individual **objectIds** and not to the complete set. The response has the **object**, **compactParameter** and **lockStatus**. Returning the **compactParameter** requires any compacted object bytes to be de-compacted before being returned as **decompactedData (15)**. Also, any bytes encoded as **applicationDefined (0)** or **utf8Data (2)** are returned with that same status. Returning the **lockStatus** requires processing by the Tag Driver to identify from RF tag features that the relevant blocks are locked.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

checkDuplicate (see 9.2.3)
completionCode (see 9.3)
executionCode (see 9.4)
lockStatus (see 9.2.5)

9.1.16 Read First Object command

The command (see 8.20 of ISO/IEC 15961) instructs the interrogator to return an **object** associated with an **objectId** from the first ordinal sequence on the Logical Memory Map of a particular RF tag. Processing the command requires calculating the number of blocks to be read from the following:

A = **maxAppLength** provides the maximum number of bytes for the expected object.

B = **objectId** converts to OBJECT IDENTIFIER or RELATIVE-OID depending on the data format, from which an encoded length can be determined.

C = 1 byte for the encodation of the Precursor.

The number of blocks to read is given by (A + B + C)/block size, and rounded up.

The response has the **objectId, object, compactParameter** and **lockStatus**. Returning the **compactParameter** requires any compacted object bytes to be de-compacted before being returned as **decompactedData (15)**. Also, any bytes encoded as **applicationDefined (0)** or **utf8Data (2)** are returned with that same status. Returning the **lockStatus** requires processing by the Tag Driver to identify from RF tag features that the relevant blocks are locked.

The following elementNames in the abstract syntax require processing as defined elsewhere in this International Standard.

completionCode (see9.3)
executionCode (see 9.4)
maxAppLength (see 9.2.6)

9.2 Processing arguments

The following arguments are required to process the application commands.

9.2.1 afiLock

The application command argument **afiLock** is Boolean. If set to TRUE the **applicationFamilyId** shall be written to the RF tag's system information and be permanently locked. If set to FALSE it shall be written but not locked.

Possible failures to complete the command argument result in the following error, represented by the **completionCodes** (more fully defined in 9.4):

afiNotConfigured (1)
afiNotConfiguredLocked (2)
afiConfiguredLockFailed (3)

9.2.2 avoidDuplicate

The application command argument **avoidDuplicate** is Boolean. If set to TRUE and the **accessMethod = directory**, the directory first shall be read into the Logical Memory and checked to confirm that the specified **objectId** is not present. If set to TRUE and the **accessMethod = noDirectory**, the data sets of Precursor, **objectId** and **object** shall be read into the Logical Memory until a duplicate **objectId** is found, or until the end

of the data sets. If the specified **objectId** is found, the write process shall be aborted and the appropriate **completionCode** used to indicate the error.

If set to FALSE, no attempt shall be made to check for duplicates.

Possible failures to complete the command argument result in the following errors, represented by the **completionCodes**:

objectNotAdded (9)
duplicateObject (10)

9.2.3 checkDuplicate

The application command argument **checkDuplicate** is Boolean. If set to TRUE and the **accessMethod = directory**, the directory first shall be read into the Logical Memory and checked to confirm that the specified **objectId** is not present. If set to TRUE and the **accessMethod = noDirectory**, the data sets of Precursor, **objectId** and **object** shall be read into the Logical Memory until a duplicate **objectId** is found, or until the end of the data sets. If set to FALSE the command argument shall action the first data set containing the **objectId**.

Possible failures to complete the command argument result in the following errors, represented by the **completionCodes**:

duplicateObject (10)
objectIdNotFound (13)

9.2.4 identifyMethod and numberOfTags

The application command argument **identifyMethod** is used to define whether all or some of the RF tags belonging to the selected **applicationFamilyId** in the operating area shall be identified. It shall be used in conjunction with the command argument **numberOfTags** (defined later in this sub-clause). The value of **identifyMethod** is an integer and the following codes apply:

0 **inventoryAllTags** - This calls for the **tagId** of all RF tags in the operating field to be identified, probably with the additional system constraint of a time limit.

1 **inventoryAtLeast** - This calls for the **tagId** of all RF tags in the operating field to be identified, but differs from (1) in that an error occurs if the minimum number is not found. The minimum quantity should reflect some value relevant to the application.

2 **inventoryNoMoreThan** - This calls for the **tagId** of a maximum number of RF tags in the operating field to be identified. This could be used for sampling purposes.

3 **inventoryExactly** - This calls for the **tagId** of a specific number of RF tags in the operating field to be identified. This argument can be used to confirm that particular items are present in the operating field (e.g. within a container).

The application command argument **numberOfTags** is a conditional qualifier to the basis argument **identifyMethod**. This numeric selection is defined by the application to confirm by an actual count, or a partial count, which particular RF tags are in the operating field. The Data Protocol Processor and the Tag Driver do not need to be concerned with the application logic of this command. However, there are some systems applications to consider:

- If the **identifyMethod** is set to **invetoryAllTags (0)**¸ the **numberOfTags** should be set to 0 in the application argument.

- If the **identifyMethod** is set to **inventoryAtLeast (1)** and the **numberOfTags** is set to 1, the RF interrogator is effectively being set to wait for an RF tag to enter the operating field. It then responds with the **tagId** of the first RF tag that it detects, and others that enter the operating area at the same time.

- Some of the identify arguments can result in an open-ended loop and an indefinite wait. Additional systems constraints may be required to respond with an appropriate **completionCode**.

9.2.5 lockStatus

The application command response argument **lockStatus** is Boolean. It requires the Tag Driver to identify features (that are not part of the Logical Memory Map nor the system information) from the RF tag that relevant blocks are locked. The response requires this information to be associated with each individual set of **objectId** and **object**. If the set is locked, the response shall be TRUE. If not locked, the response shall be FALSE.

As the intention is to identify the status of what is actually encoded on the RF tag, no errors are associated with this process.

9.2.6 maxAppLength

The application command argument **maxAppLength** is used to define the length in bytes of the **object** associated with an **objectId**, in the first position stored on the RF tag. The value of **maxAppLength** is an integer. This value is used as part of a calculation to determine the number of blocks to be read (see 9.1.16).

Possible failures to complete the command argument result in the following error, represented by the **completionCode**:

objectNotFound (13)

9.2.7 objectLock

The application command argument **objectLock** is Boolean. If set to TRUE the data set of Precursor, **objectId** and **object** shall be locked, but the **objectId** shall not be locked within the directory, if a directory is used. The locking process needs to be effected within block boundaries. Therefore, a locked data set shall not overlap blocks that contain data sets that are not to be locked. The resultant re-adjustment of logical addresses could most probably require the use of an offset in the data set, signalled in the Precursor. This can apply to the data set directly associated with the command argument, if subsequent data sets are either not to be locked or not defined by the command. Also, if this preceding data set does not end on a block boundary, it shall require an offset to end on a block boundary. For dataFormat 2, if the root-OID is not already locked then any additional object required to be locked shall be encoded with its full OBJECT IDENTIFIER and the complete data shall be locked.

Possible failures to complete the command argument result in the following errors, represented by the **completionCodes**:

objectAddedButNotLocked (11)
objectModifiedButNotLocked (22)

NOTE: The **completionCodes** listed above apply selectively to particular commands.

9.2.8 storageFormatLock

The application command argument **storageFormatLock** is Boolean. If set to TRUE the **storageFormat** shall be written to the RF tag's system information and be permanently locked. If set to FALSE it shall be written but not locked.

Possible failures to complete the command argument result in the following errors, represented by the **completionCodes**:

storageFormatNotConfigured (4)
storageFormatNotConfiguredLocked (5)
storageFormatConfiguredLockFailed (6)

9.3 Completion codes

If a command fails to be properly executed and an error is identified, the response to the application needs to include this information. This is achieved by means of the appropriate **completionCode** as specified below. As only one **completionCode** can be returned per response (because of the structure of the transfer syntax), the first error that is encountered shall determine the value of the code. Once an error condition has been identified, the RF tag interrogator may choose to end processing and close the command. Addressing multiple errors is beyond the scope of this international Standard, except that the Read Logical Memory Map command can be used for diagnostic purposes.

noError (0)

afiNotConfigured (1)
For some reason, possibly because of a prior configure action, the command was not completed.

afiNotConfiguredLocked (2)
As (1) but responding that the existing **applicationFamilyId** was also found to be locked.

afiConfiguredLockFailed (3)
Successfully configured but not locked as required.

storageFormatNotConfigured (4)
For some reason, possibly because of a prior configure action, the command was not completed.

storageFormatNotConfiguredLocked (5)
As (4) but responding that a pre-existing **storageFormat** was also found to be locked.

storageFormatConfiguredLockFailed (6)
Successfully configured but not locked as required.

objectLockedCouldNotModify (7)
The previous encodation on the RF tag was locked, and as a result the attempt to update the **object** value could not be completed.

tagIdNotFound (8)
The particular RF tag, specified by the **tagId**, could not be found in the operating area.

objectNotAdded (9)
The data set of Precursor, **object** and **objectId** could not be added to the Logical Memory Map (e.g. because there was insufficient memory space). This response also applies if the RF tag supports a lock function that failed.

duplicateObject (10)
An **objectId** with the same value as the one to be processed (added, modified, or deleted) was found. The process was aborted.

objectAddedButNotLocked (11)
The data set was added to a RF tag that did not support a lock feature in the memory.

objectNotDeleted (12)
The data set of Precursor, **objectId** and **object** could not be deleted from the Logical Memory Map (e.g. because the delete function is not supported by the particular RF tag).

objectIdNotFound (13)
The **objectId** intended to be processed was not actually encoded on the RF tag.

objectLockedCouldNotDelete (14)
The data set of Precursor, **objectId** and **object** is locked and could not be deleted.

objectNotRead (15)
The intention to read the specified object failed.

objectsNotRead (16)
The intention to read the specified object failed.

blocksLocked (17)
This indicates that the intended action to erase encoded bytes from one or more blocks could not be actioned.

eraseIncomplete (18)
This indicates that the intended action to erase encoded bytes from one or more blocks was interrupted and that not all blocks have been processed The command can be re-invoked to complete the process.

readIncomplete (19)
The intention to read the complete contents of the Logical Memory Map failed, because not all blocks from the RF tag were transferred to the Logical Memory.

systemInfoNotRead (20)
The intention to read the system information failed.

objectNotModified (21)
The data set of Precursor, **objectId** and **object** could not be modified on the Logical Memory Map (e.g. because the modify function is not supported by the particular RF tag).

objectModifiedButNotLocked (22)
The **object** was modified in a RF tag that did not support a lock feature in the memory.

failedToReadMinimumNumberOfTags (23)
The minimum number of RF tags were not identified with the specified selection criterion, possibly because of a time-out.

failedToReadExactNumberOfTags (24)
The pre-defined exact number of RF tags were not identified with the specified selection criterion, possibly because of a time-out.

undefinedCommandError (254)

An error occurred in a command, not defined by another code.

executionError (255)
A system error occurred which made it impossible to action the command. The appropriate **executionCode** is returned in the response.

9.4 Execution codes

If a command fails to be completely executed because of a systems error, the response to the application needs to include this information. This is achieved by means of the appropriate **executionCode** as specified in Clause 8.3 of ISO/IEC 15961. As only one **executionCode** can be returned per response (because of the structure of the transfer syntax), the first system error that is encountered shall determine the value of the code.

10 Communications between the Data Protocol and the RF tag

ISO/IEC 15961 presents a standardised way for application commands and data to be exchanged with the application using the Data Protocol. This International Standard also provides standard processes to convert those application commands and data into a byte stream to be encoded on the RF tag and to provide the reverse process for reading from the RF tag. This standardisation is possible because RFID for item management, particularly in open systems, has yet to be implemented on a large scale basis at the initial publication of these standards.

In contrast, RF tag architectures have been evolving as artefacts up to the point of standardisation in ISO/IEC 18000. To accommodate the various architectures and RF commands a Tag Driver specific to an ISO/IEC 18000 mode shall act as the interface between the Data Protocol and the RF tag commands and processes.

The Tag Driver has the following key functions:

- It provides the means of transferring the system information between the Data Protocol Processor and the RF tag, and vice versa.

- It defines the mapping rules for converting the Logical Memory in the Data Protocol Processor to the Logical Memory Map on the RF tag(s) with which communications are being maintained. It also supports the reverse function of transferring from the RF tag to the Logical Memory in the Data Protocol Processor.

- It facilitates the conversion of application commands into command codes supported by the particular RF tag. It also converts the specific RF tag format of responses to the common framework of the Data Protocol.

  NOTE: The feature to address application commands and responses by the Tag Driver is not explicitly addressed in this International Standard. This is subject to specific implementation.

- If the RF tag supports a mechanism to lock blocks of memory, the Tag Driver provides a means of transferring to the Data Protocol processor the information that a particular block, or group of blocks, is locked.

Annex A provides a pro forma of the Tag Driver definition. Annex B provides the details of the Tag Driver for each mode of ISO/IEC 18000.

11 Compliance, or classes of compliance, to this International Standard

11.1 Compliance of the Data Protocol Processor

For a Data Protocol Processor to claim compliance with this International Standard, it shall support all of the commands and their constituent parts.

The Data Protocol Processor may support, selectively, one or more of the Tag Drivers defined in Annex B.

11.2 Compliance of the Tag Driver

A Tag Driver shall provide all the requirements defined for the particular air interface as detailed in Annex B.

With reference to application commands and responses, a Tag Driver shall provide support compliant with Annex A.5. For whatever reason, if a device only supports selective command and response modules, then compliance can be claimed for these particular modules by responding with the **executionCode = commandNotSupported (4)**. If the claim is made to support a command, then every constituent feature of that command shall be supported.

Annex A
(normative)

Pro Forma Description for the Tag Driver

This annex provides the structure for defining a Tag Driver description for a class of RF tags compliant with a mode of ISO/IEC 18000. The details in Annex B, provide more specific examples of the type of definition required.

A.1 Defining the tagId

The **tagId** is the device in the Data Protocol to link the RF tag to the Logical Memory. Clause 7.2.1 identifies some of the possible methods that can be invoked on the RF tag, the air interface, or even using unique encoded data to achieve this.

The Tag Driver shall define the technique that the RF tag uses to create the **tagId**.

A.2 System information : applicationFamilyId

The **applicationFamilyId** (see 7.2.4) is a prime selection mechanism for specifying a relevant subset of RF tags that could be in the operating area of the interrogator. Its coded value is one byte long and its value shall be defined by the application.

The Tag Driver shall define whether the RF tags supports the **applicationFamilyId** feature, not just as a provision for encoding the value, but also for it to be used as a selection tool. The Tag Driver shall also specify whether the code value, once encoded, can be locked and if this locking facility is mandatory or optional.

A.3 System information : storageFormat

The **storageFormat** (see 7.2.5) uses the **accessMethod** to define the structure of the bytes on the RF tag's Logical Memory Map, and uses the **dataFormat** to define particular sets of encoded application data. It is a key to the Data Protocol Processor on how to organise selective transactions defined by the application commands and how to condense the **objectIds** during encoding and to expand it during decoding. The **storageFormat** is one byte long and its value shall be defined by the application.

The Tag Driver shall define whether the RF tags support the **storageFormat** feature, not just as a provision for encoding the value, but also for it to be used as an organisational tool, particularly for how the different **accessMethods** are supported. The Tag Driver shall also specify whether the code value, once encoded, can be locked and if this locking facility is mandatory or optional.

A.4 Memory-related parameters

A.4.1 Block size

The Tag Driver shall specify the size of a block in terms of number of bytes and the means of transferring this information to the Data Protocol Processor.

A.4.2 Locking feature

The Tag Driver shall specify whether any, all, or some of the blocks can be selectively locked to render it impossible (or at least extremely difficult) to change the encoded bytes in a block.

A.4.3 Number of blocks

The Tag Driver shall specify the number of blocks on the particular RF tag and the means of transferring this information to the Data Protocol Processor.

A.4.4 Memory mapping

The Tag Driver shall specify:

1. The location within the user memory (as defined in ISO/IEC 18000) where the Logical Memory Map begins.

2. The byte sequence in the block - the Data Protocol presumes Most Significant Byte in the first position, so a different order may require a conversion process.

3. The bit sequence in the byte - the Data Protocol presumes most significant bit in the first position, so a different ordering requires a conversion process.

4. Any other structuring rules within the ISO/IEC 18000 definition of user memory (e.g. error correction bits or bytes) that impact on the structure of the Logical Memory Map.

   NOTE: If error correction consumes bit or byte capacity outside the block size defined for user data, then this can be ignored for the purposes of the Data Protocol Processor.

A.5 Support for commands

The manner in which the application commands are presented in ISO/IEC 15961 is likely to be at a higher level than the coded commands communicated across the air interface. Each application command shall be supported in one of the following ways:

a. fully support (i.e. action) the functionality across the air interface.

b. fully support the functionality of some of the command arguments and responses, but return appropriate error codes for those arguments not supported. For example, a command argument of **objectLock** might not be supported by the RF tag command calls; therefore the appropriate **completionCodes** shall be used.

c. be unable to support the functionality of a complete application command, for example to modify or delete an **objectId** and **object** on an RF tag that is One Time Programmable (or that uses Write Once Read Many technology).

The Tag Driver shall identify in broad terms how the two levels of command (application command and RF tag command) are linked. Precise coding rules are not required and are to be resolved at the implementation level.

Annex B
(normative)

ISO/IEC 18000 Tag Driver Descriptions

B.1 Tag Driver for ISO/IEC 18000-2: Parameters for Air Interface Communications below 135 kHz

This normative annex explains and specifies how this International Standard shall be implemented on tags compliant to ISO/IEC 18000-2.

B.1.1 Defining the TagId

Tags are identified by a 64 bit Unique Identifier (UID). The UID is used during arbitration and for addressing each tag individually.

ISO/IEC 18000-2 supports commands that can ignore the UID, such that all tags in the ready state shall execute the command.

B.1.2 System information : applicationFamilyId

AFI (Application Family Identifier) represents the type of application targeted by the Interrogator and is used to extract from all the tags present only those tags meeting the required application criterion. AFI may be programmed and locked by the respective commands. AFI is coded on one byte, which constitutes 2 nibbles of 4 bits each.

The support of AFI by the tag is optional. If AFI is not supported by the tag and if the AFI flag is set in a command, the tag shall not respond whatever the AFI value is in the request.

If the AFI mechanism is supported by the tag, it shall respond when the AFI on the tag matches the AFI in the request; or if the command specifies AFI 00_{HEX} irrespective of the AFI value on the tag. The tag shall also respond if the AFI flag is not set in the command.

B.1.3 System information : storageFormat

StorageFormat is supported in ISO/IEC 18000-2 through the DSFID mechanism. The DSFID is on one byte. DSFID is returned during the Inventory process. The DSFID can be programmed and locked by specific commands. If the tag does not support DSFID programming, the tag shall return DSFID='00'.

B.1.4 Memory-related parameters

The physical tag memory is divided into two logical sections. The first logical memory section contains the system data. The second logical memory section contains the user data and is referred to as the application memory.

The user data or application memory is organised in blocks of a fixed number of bytes. Up to 64 blocks can be addressed, starting from block 0. Block sizes can be 4, 8, 12 or 16 bytes. Each block may be locked independently of the others, using a bitwise one time programmable 64 bit field in the system data. Information on the number of blocks and the size of the blocks is returned by the Get System Information command.

B.1.5 Support for commands

Not all commands are mandatory. If a tag does not support a command, the tag shall answer with an error code as specified in ISO/IEC 18000-2. This allows the application and/or the interrogator to know that they cannot use the requested feature on this tag.

B.2 Tag Driver for Mode 1 of ISO/IEC 18000-3: Parameters for Air Interface Communications at 13,56 MHz

This normative annex explains and specifies how this International Standard shall be implemented on tags compliant to ISO/IEC 18000-3 Mode 1.

B.2.1 Defining the TagId

Each tag is identified uniquely by a UID on 64 bits. Each tag returns its UID during the inventory process. The UID can be further used to selectively address a tag. The UID format is specified in ISO/IEC 18000-3 Mode 1.

B.2.2 System information : applicationFamilyId

The applicationFamilyId is supported in ISO/IEC 18000-3 Mode 1 through the AFI mechanism. The AFI can be programmed and locked by specific commands. The Inventory process then uses it. The AFI is on 8 bits.

If the tag does not support AFI programming (see B.2.5), the tag shall participate to the Inventory process as if it had received AFI='00', meaning that the tag will always respond to an Inventory command, whatever the AFI value.

B.2.3 System information : storageFormat

StorageFormat is supported in ISO/IEC 18000-3 Mode 1 through the DSFID mechanism. DSFID is returned during the Inventory process. The DSFID can be programmed and locked by specific commands. The DSFID is on 8 bits.

If the tag does not support DSFID programming (see B.2.5), the tag shall return DSFID='00'.

B.2.4 Memory-related parameters

The System Information is stored in a memory area that is logically distinct from the User memory and that can be accessed only by specific commands. Information on the number of blocks and the size of the blocks is returned by the Get System Information command.

The user memory is organised in blocks, starting at block 0. The logical memory mapping shall start at block 0. Bits and bytes ordering shall be the same, i.e. the msb and MSB shall match in both the logical memory and in the tag memory.

ISO/IEC 18000-3 Mode 1 supports the selective locking of blocks. The hardware implementation is determined by the manufacturer of the chip.

B.2.5 Support for commands

Only two commands are mandatory, the Inventory command and the Quiet command. All other commands are optional. Some tag products may therefore not have implemented them. If a tag does not support a command, the tag shall answer with an error code as specified in ISO/IEC 18000-3 Mode 1. This allows the application and/or the interrogator to know that they cannot use the requested feature on this tag.

B.2.6 Performance optimisation

If the application and/or the interrogator plan to have a continuous sequence of commands with a given tag, they can use the Select command. The Select command is addressed to a specific tag and deselects all other tags. It contains the UID of the tag to be selected. All further commands shall have their Select flag set, but do not contain the tag UID, therefore saving transmission time. Individual commands with other UIDs can be sent to specific tags during the sequence. See ISO/IEC 18000-3 Mode 1 for details.

B.3 Tag Driver for Mode 2 of ISO/IEC 18000-3: Parameters for Air Interface Communications at 13,56 MHz

This normative annex explains and specifies how this International Standard shall be implemented on tags compliant to ISO/IEC 18000-3 Mode 2.

B.3.1 Defining the TagId

Each tag is identified uniquely by a Specific Identifier (SID). Each tag returns its SID during the Application Group Identifier command. Subsequent commands may include the SID to address a specific tag.

B.3.2 System information : applicationFamilyId

The applicationFamilyId is supported in ISO/IEC 18000-3 Mode 2 through the Application Group Identifier (GID) command. Tags will respond to valid commands if the GID in the command is equal to the GID stored in the tag memory. The tags will also respond if the GID in the command is set to $FFFF_{HEX}$.

B.3.3 System information : storageFormat

The protocol describes Hardcode fields that provide a function similar to system information, and include the Storage Format. If requested by reader command the Hardcode fields are included in Tag replies. The protocol describes the Hardcode in virtual terms only. Thus the Hardcode may be realised by mask or memory. If mask is chosen the Hardcode is set at the chip design. If memory is chosen the Hardcode is written using a special write to memory.

B.3.4 Memory-related parameters

The tag memory is organised and addressed as 16-bit words. ISO/IEC 18000-3 Mode 2 supports tag types with varying block sizes, where a block is one or more 16-bit words. The total memory capacity is defined in words, so the number of blocks can be derived from this.

The user memory is addressed from Word 0 upwards. Read commands address the memory on word boundaries. Write commands address the memory on block boundaries. The protocol allows commands to address user memory using 8-bit address and 8-bit length or 16-bit address and 16-bit length.

The user memory may be locked by using a locked pointer. All memory addresses less than the value stored in the lock pointer are locked addresses. The lock pointer value can only be incremented. The lock pointer is updated using a special write command.

B.3.5 Support for commands

The tag will respond to valid commands. No response is sent for an invalid command. Invalid commands include: commands with invalid command types, invalid identifiers, invalid address ranges, attempts to write to locked memory or invalid CRCs.

Tags respond to valid read commands by providing the requested data. Tags respond to valid write commands by writing the data included in the command to the tag memory and then responding to the write

command. The write response can include read data if requested by the write command (read/write command).

B.4 Tag Driver for ISO/IEC 18000-4: Parameters for Air Interface Communications at 2,45 GHz - Mode 1

This normative annex explains and specifies how this International Standard shall be implemented on tags compliant to ISO/IEC 18000-4 Mode 1.

B.4.1 Defining the TagId

Each tag is identified uniquely by a UID of 64 bits. Each tag returns its UID during the inventory process. The UID can be further used to address selectively a tag for subsequent transactions.

B.4.2 System information : applicationFamilyId

The applicationFamilyId is supported in ISO/IEC 18000-4 Mode 1 through the AFI mechanism. Bytes 12 through 17 of tag system information are reserved for tag memory system information. Byte 12 represents the Embedded Application Code (EAC) field. This information field defines the data architecture system represented in application memory (bytes 17 and above). The remaining five (5) bytes define the memory architecture and usage within the data architecture system defined in byte 12. If this byte is set to the value $0A_{HEX}$ the tag data architecture is compliant to this International Standard. The data field termed Application Family Identifier (AFI) is represented in the next byte (byte 13) of the reserved tag memory system information.

If the tag does not support AFI programming, the tag shall participate to the Inventory process as if it had received AFI='00', meaning that the tag will always respond to an Inventory command, whatever the AFI value.

B.4.3 System information : storageFormat

StorageFormat is supported in ISO/IEC 18000-4 Mode 1 through the DSFID mechanism. DSFID indicates how the application data is structured in the tag application memory. As defined in B.4.2, bytes 12 through 17 are reserved in tag system memory for representation of tag memory system information. Byte 12 with the value $0A_{HEX}$ defines a tag compliant with this International Standard. Byte 14 is designated to store the DSFID information. As this information can be stored in tag system memory, read (READ) and write (WRITE) commands may be used to program and retrieve this information.

If the tag does not support DSFID programming, the tag shall return DSFID='00'.

B.4.4 Memory-related parameters

The first 18 octets (bytes 0 through 17) are reserved for system information. The Identification process provides information about the tag from reserved memory locations. Upon tag segregation (the anti-collision process), the tag state is moved from the “Identify” state to the “Data Exchange” state with the retrieval of such relevant tag system information.

Bytes 12, 13, and 14 are as defined above. Bytes 10 and 11 are reserved for “Hardware Tag Type”. This two-byte field provides information about the hardware (physical) tag, which includes the total number of blocks and block size (bytes per block).

The application memory (user data storage) begins in byte 18 (or the first addressable block after system data). This is treated as the first byte of block 0 of the logical memory map for user memory. Bits and bytes ordering shall be the same, i.e. the msb and MSB shall match in both the logical memory and in the tag memory.

As noted in the response to question three above, unique and unambiguous addressing of all data transactions ia provided through the Tag UID (bytes 0 through 7 of system memory). All data transactions are uniquely addressed with this mechanism. The submission provides

ISO/IEC 18000-4 Mode 1 supports writing and locking of data with two commands: (WRITE) and (LOCK). These commands operate at the block level. The locking mechanism is physical (fusible link) and protects data from any change permanently. Once the data block is “locked” it cannot be “unlocked”. All data that is locked cannot be changed through the air interface.

B.4.5 Support for commands

Not all commands are mandatory. If a tag does not support a command, the tag shall answer with an error code as specified in ISO/IEC 18000-4 Mode 1. This allows the application and/or the interrogator to know that they cannot use the requested feature on this tag.

B.5 Tag Driver for ISO/IEC 18000-4: Parameters for Air Interface Communications at 2,45 GHz - Mode 2

This normative annex explains and specifies how this International Standard shall be implemented on tags compliant to ISO/IEC 18000-4 Mode 2. This mode supports different user interfaces and the following sub-clauses are based on the implementation using the “real image” user interface.

B.5.1 Defining the TagId

Each tag is identified uniquely a tag ID of 32 bits. The tag ID is used to address selectively a tag for subsequent transactions.

B.5.2 System information : applicationFamilyId

ISO/IEC 18000-4 Mode 2 does not support the functionality of applicationFamilyId. As the Mode does not support AFI programming, the tag shall participate in any application command that specifies applicationFamilyId as if it had received AFI='00', meaning that the tag will always respond to an applicationFamilyId argument in an application command, whatever the AFI value.

B.5.3 System information : storageFormat

ISO/IEC 18000-4 Mode 2 does not support the functionality of storageFormat. This has the effect of reducing support for this International Standard to:

- AccessMethod = noDirectory
- StorageFormat = fullFeature

B.5.4 Memory-related parameters

The tag ID provides supplementary information about the size of the user memory. Blocks are always one byte in length, irrespective of the size of memory. Read or write functions address memory by using a 2-byte position that identifies the beginning of the required data string, and a single byte value to determine its length. The maximum length of the data string is 246 bytes.

B.5.5 Support for commands

ISO/IEC 18000-4 Mode 2 supports a number of commands including read and write. The response to the command includes specific error codes where the command could not be executed.

B.6 Tag Driver for ISO/IEC 18000-6: Parameters for Air Interface Communications at 860 MHz to 960 MHz

This normative annex explains and specifies how this International Standard shall be implemented on tags compliant to ISO/IEC 18000-6.

B.6.1 Defining the TagId

Each tag is identified uniquely by a UID of 64 bits (Type A and B) or an SUID of 40 bits (Type A). Each tag returns its UID (or SUID) during the inventory process. The UID/SUID can be further used to address selectively a tag for subsequent transactions.

The UID format is specified in ISO/IEC 18000-6 in the following places:

- Type A is defined in clause 8.1.2
- Type B is defined in clause B.2

B.6.2 System information : applicationFamilyId

The applicationFamilyId is supported in ISO/IEC 18000-6 through the AFI mechanism.

For ISO/IEC 18000-6 Type A: the AFI can be programmed and locked by specific commands. The Inventory process then uses these commands to retrieve the AFI data. The AFI is on 8 bits.

For ISO/IEC 18000-6 Type B: bytes 12 through 17 of tag system information are reserved for tag memory system information. Byte 12 represents the Embedded Application Code (EAC) field. This information field defines the data architecture system represented in application memory (bytes 17 and above). The remaining five (5) bytes define the memory architecture and usage within the data architecture system defined in byte 12. If this byte is set to the value $0A_{HEX}$ the tag data architecture is compliant to this International Standard. The data field termed Application Family Identifier (AFI) is represented in the next byte (byte 13) of the reserved tag memory system information.

If the tag (Type A or B) does not support AFI programming (see B.6.5), the tag shall participate to the Inventory process as if it had received AFI='00', meaning that the tag will always respond to an Inventory command, whatever the AFI value.

B.6.3 System information : storageFormat

StorageFormat is supported in ISO/IEC 18000-6 through the DSFID mechanism.

For ISO/IEC 18000-6 Type A: DSFID is returned during the Inventory process. The DSFID can be programmed and locked by specific commands. The DSFID is defined on 8 bits.

For ISO/IEC 18000-6 Type B: DSFID indicates how the application data is structured in the tag application memory. As defined in B.6.2, bytes 12 through 17 are reserved in tag system memory for representation of tag memory system information. Byte 12 with the value $0A_{HEX}$ defines a tag compliant with this International Standard. Byte 14 is designated to store the DSFID information. As this information can be stored in tag system memory, read (READ) and write (WRITE) commands may be used to program and retrieve this information.

If the tag (Type A or B) does not support DSFID programming (see B.6.5), the tag shall return DSFID='00'.

B.6.4 Memory-related parameters

B.6.4.1 User memory for ISO/IEC 18000-6 Type A

The System Information is stored in a memory area that is logically distinct from the user memory and that can be accessed only by specific commands. Information on the number of blocks and the size of the blocks is returned by the Get System Information command.

The logical memory mapping for user memory shall start at block 0. Bits and bytes ordering shall be the same, i.e. the msb and MSB shall match in both the logical memory and in the tag memory.

B.6.4.2 User memory for ISO/IEC 18000-6 Type B

The first 18 octets (bytes 0 through 17) are reserved for system information. The Identification process provides information about the tag from reserved memory locations. Upon tag segregation (the anti-collision process), the tag state is moved from the “Identify” state to the “Data Exchange” state with the retrieval of such relevant tag system information.

Bytes 12, 13, and 14 are as defined above. Bytes 10 and 11 are reserved for “Hardware Tag Type”. This two-byte field provides information about the hardware (physical) tag, which includes the total number of blocks and block size (bytes per block).

The application memory (user data storage) begins in byte 18 (or the first addressable block after system data). This is treated as the first byte of block 0 of the logical memory map for user memory. Bits and bytes ordering shall be the same, i.e. the msb and MSB shall match in both the logical memory and in the tag memory.

B.6.5 Support for commands

ISO/IEC 18000-6 provides for both mandatory and optional commands. Mandatory commands provide the minimum functionality for activation and identification (e.g. inventory) of compliant tags in a population. All other commands (reading and writing) are optional. Some tag products may therefore not have implemented them. If a tag does not support a command, the tag shall answer with an error code as specified in ISO/IEC 18000-6. This allows the application and/or the interrogator to know that they cannot use the requested feature on this tag.

B.6.6 Performance optimisation

If the application and/or the interrogator requires a continuous dialogue with a specific tag, ISO/IEC 18000-6 provides a mechanism for "selection" through the command set.

For ISO/IEC 18000-6 Type A, the Select command is addressed to a specific tag and deselects all other tags. It contains the UID of the tag to be selected. All further commands shall have their Select flag set, but do not contain the tag UID, therefore saving transmission time. Individual commands with other UIDs can be sent to specific tags during the sequence.

For ISO/IEC 18000-6 Type B, the group selection commands (GROUP_SELECT_XX) provide an efficient means to logically query the contents of either system memory (e.g. UID, ASF, DSFID, etc.) or user memory as a means to for selecting a subset of tags for data transactions. Only the tags logically queried as TRUE based on the contents of the selected memory (system and/or user) would thus be activated and participate in subsequent data transactions.

Annex C
(normative)

Data Compaction Schemes

The data compaction schemes are applicable to data objects to reduce the amount of encoding space required to store that data on the RF tag. The schemes shall apply to entire data objects, i.e. it is not possible to switch schemes in the middle of a data object. Nor shall a compaction scheme straddle two, or more, data objects. By applying data compaction to a complete object, it can be extracted in its compacted form as part of a read or write command.

The schemes are defined below in sequence of greatest potential compaction to no compaction.

C.1 Integer compaction

Integer compaction is designed to compact decimal integers from the value 10 to 9999999999999999999 (i.e. any 2-digit to 19-digit value) to a binary format. All input bytes shall be in the range 30 to 39_{HEX}; and the leading byte(s) shall not be 30_{HEX}.

If the decimal integer value is less than 10, or is longer than 19 digits, or the leading byte(s) are $30_{HEX,}$ numeric compaction shall be applied.

The rules for integer compaction are:

1. If the decimal numeric value is 10 to 9999999999999999999, convert to a binary value.
   NOTE: This allows for conversion within a 64-bit value (or 8 bytes). Some program languages able to support a simple data type conversion to an integer value (different names are used). If the particular language does not support a data type conversion of a decimal value of 19 digits, then a two-stage process should be used:
   a. Use the data type conversion up to the limit of the program language
   b. Use a Modulo 256 conversion for higher values

2. Align to a byte boundary, by padding with leading zero bits if required. Depending on the conversion procedure used, it could be necessary to strip off any leading bytes with the value 00_{HEX} to achieve the minimum encoded length. The encoded byte string should not include Encode as integer**,** code value 001 in the Precursor.

C.2 Numeric compaction

Numeric compaction is designed to encode any decimal numeric character string, including leading zeros. The character string shall be 2 or more characters long. Numeric compaction preserves the original character string length so that, once decoded, leading zeros, if present, are output. All input bytes shall be in the range 30 to 39_{HEX}.

The rules for numeric compaction are:

1. Convert each decimal digit to its 4-bit binary equivalent (Binary Coded Decimal).

2. If the numeric character string has an odd number of digits, append an additional 4-bit string “1111” to align the compaction to byte boundaries.

3. Encode each 4-bit pair as an byte. Define the compacted sequence as numeric**,** code value 010 in the Precursor.

During the decode process, if the last byte has the value “xF”, the last four bits “1111” are discarded to re-create the numeric character string of an odd number of decimal digits.

C.3 5-bit compaction

5-bit compaction is designed to encode uppercase Latin characters and some punctuation. All input bytes shall be in the range 41 to $5F_{HEX.}$ The character string shall be 3 or more characters long. Up to 37% of memory space can be saved using this scheme. Annex D shows the ISO/IEC 646 characters that can be encoded.

The rules for 5-bit compaction are:

1. For each character:

    a. Confirm that the byte value is in the range 41 to $5F_{HEX}$.
    b. Convert the byte value to its 8-bit binary equivalent.
    c. Strip off the lead 3 bits “010”.
    d. Write the remaining 5-bits to a bit string.

2. Once all the characters have been converted to 5-bit values and concatenated, divide the resultant bit string into 8-bit segments starting with the most significant bit. If the last segment contains less than 8 bits, pad with “0” bits.

3. Convert the 8-bit segments to hexadecimal values.

4. Encode the converted byte sequence as 5 bit code, code value 011 in the Precursor.

During the decode process, each 5-bit segment of the compacted bit string has “010” added as a prefix to re-create the 8-bit value of the source data. If “0” pad bits are present at the end of the compaction bit string, they are discarded.

If 5, 6 or 7 pad bits are present, the decoder could attempt to convert the first 5-bits to the source data. However, this results in character 40_{HEX}, which is not supported in 5-bit compaction and shall be discarded.

C.4 6-bit compaction

6-bit compaction is designed to encode uppercase Latin characters, numeric digits and some punctuation. All input bytes shall be in the range 20 to $5F_{HEX}$. If the trailing byte(s) are 20_{HEX}, 7-bit compaction shall be used. The character string shall be 4 or more characters long. Up to 25% of memory space can be saved using this scheme. Annex D shows the ISO/IEC 646 characters that can be encoded.

The rules for 6-bit compaction are:

1. Check for byte 20_{HEX} in the final position(s). If found, go to 7-bit compaction, otherwise continue steps 2 to 5.

2. For each character:

    a. Confirm that the byte value is in the range 20 to $5F_{HEX.}$
    b. Convert the byte value to its 8-bit binary equivalent.
    c. Strip off the leading 2 bits: “00” for bytes 20 to $3F_{HEX}$ or “01” for bytes 40 to $5F_{HEX}$.
    d. Concatenate the remaining 6-bits to a bit string.

3. Divide the resultant bit string into 8-bit segments starting from the most significant bit. If the last segment contains less than 8 bits pad, as appropriate, with the first two, four or all bits of the pad string “100000”.

4. Convert the 8-bit segments to hexadecimal values.

5. Encode the converted byte sequence as 6 bit code, code value 100 in the Precursor.

During the decode process, each 6-bit segment of the compacted bit string is analysed.

a. If the first bit is “1”, the bits “00” are added as a prefix before converting to values 20 to $3F_{HEX}$.

b. If the first bit is “0”, the bits “01” are added as a prefix before converting to values 40 to $5F_{HEX}$.

If pad strings “10”, “1000” or “100000” are present at the end of the encoded bit string, they are discarded.

If 6 pad bits are present, the decoder could attempt to convert this to source data. This results in character 20_{HEX} that is not supported in this final position and shall be discarded.

Using the example in 8.1.2, the example below shows the effect of processing the **object** through the Data Compactor.

EXAMPLE:

The **object** content {ABC123456} converts to HEX as 41 42 43 31 32 33 34 35 36. Analysing this byte stream shows that all values are in the range 20 to $5F_{HEX}$, enabling 6-bit compaction to be used. The Object byte stream converts as follows:

HEX: 41 42 43 31 32 33 34 35 36

Binary: 10000001 10000010 10000011 00110001 00110010
00110011 00110100 00110101 00110110

Remove bits 8 & 7: 000001 000010 000011 110001 110010
110011
110100 110101 110110

As this is only 54 bits, the first two bits of the pad string "10" are appended and the 56 bit string is divided into a sequence of 8-bit values:
000001 00 0010 0000 11 110001 110010 11 0011 1101
00 110101 110110 10

Convert to HEX: 04 20 F1 CB 3D 35 DA

C.5 7-bit compaction

7-bit compaction is designed to encode all ISO/IEC 646 characters including control characters except for DELETE. All input characters shall be in the range 00 to $7E_{HEX}$. The character string shall be 8 or more characters long. Up to 12% of memory space can be saved using this scheme. Annex D shows the ISO/IEC 646 characters that can be encoded.

The rules for 7-bit compaction are:

1. For each character:

   a. Confirm that the byte value is in the range 00 to $7E_{HEX}$.
   b. Convert the byte value to its 8-bit binary equivalent.
   c. Strip off the lead bit “0”.
   d. Concatenate the remaining 7-bits to a bit string.

2. Once all the characters have been converted to 7-bit values, divide the resultant bit string into 8-bit segments starting with the most significant bit. If the last segment contains less than 8-bits, pad with “1” bits.

3. Convert the 8-bit segments to hexadecimal values.

4. Encode the converted byte sequence as 7 bit code, code value 101 in the Precursor.

During the decode process, each 7-bit segment of the compacted bit string has bit “0” added as a prefix to recreate the 8-bit value of the source data. If “1” pad bits are present at the end of the encoded bit string, they are discarded.

If 7 pad bits are present, the decoder could attempt to convert these to source data. However, this results in character $7F_{HEX}$, which is not supported in 7-bit compaction and shall be discarded.

C.6 Octet encodation

Octet encodation is used when none of the above compaction schemes can be invoked. It encodes all bytes in the range 00 to FF.

The encoded byte string is identical to the source byte string. Encode as octet string, code value 110 in the Precursor. No decode processing is required.

Annex D
(normative)

ISO/IEC 646 Characters Supported by the Compaction Schemes

ISO/IEC 646 Character	Octet Value (HEX)	Included in Compaction Type			
		7-bit	6-bit	5-bit	Numeric
NUL	00	•			
SOH	01	•			
STX	02	•			
ETX	03	•			
EOT	04	•			
ENQ	05	•			
ACK	06	•			
BEL	07	•			
BS	08	•			
HT	09	•			
LF	0A	•			
VT	0B	•			
FF	0C	•			
CR	0D	•			
SO	0E	•			
SI	0F	•			
DLE	10	•			
DC1	11	•			
DC2	12	•			
DC3	13	•			
DC4	14	•			
NAK	15	•			
SYN	16	•			
ETB	17	•			
CAN	18	•			
EM	19	•			
SUB	1A	•			
ESC	1B	•			
FS	1C	•			
GS	1D	•			
RS	1E	•			
US	1F	•			
SPACE	20	•	•		
!	21	•	•		
“	22	•	•		
#	23	•	•		
$	24	•	•		

ISO/IEC 646 Character	Octet Value (HEX)	Included in Compaction Type			
		7-bit	6-bit	5-bit	Numeric
%	25	•	•		
&	26	•	•		
‘	27	•	•		
(	28	•	•		
)	29	•	•		
*	2A	•	•		
+	2B	•	•		
,	2C	•	•		
-	2D	•	•		
.	2E	•	•		
/	2F	•	•		
0	30	•	•		•
1	31	•	•		•
2	32	•	•		•
3	33	•	•		•
4	34	•	•		•
5	35	•	•		•
6	36	•	•		•
7	37	•	•		•
8	38	•	•		•
9	39	•	•		•
:	3A	•	•		
;	3B	•	•		
<	3C	•	•		
=	3D	•	•		
>	3E	•	•		
?	3F	•	•		
@	40	•	•		
A	41	•	•	•	
B	42	•	•	•	
C	43	•	•	•	
D	44	•	•	•	
E	45	•	•	•	
F	46	•	•	•	
G	47	•	•	•	
H	48	•	•	•	
I	49	•	•	•	
J	4A	•	•	•	
K	4B	•	•	•	
L	4C	•	•	•	
M	4D	•	•	•	
N	4E	•	•	•	
O	4F	•	•	•	
P	50	•	•	•	

ISO/IEC 646 Character	Octet Value (HEX)	Included in Compaction Type			
		7-bit	6-bit	5-bit	Numeric
Q	51	•	•	•	
R	52	•	•	•	
S	53	•	•	•	
T	54	•	•	•	
U	55	•	•	•	
V	56	•	•	•	
W	57	•	•	•	
X	58	•	•	•	
Y	59	•	•	•	
Z	5A	•	•	•	
[	5B	•	•	•	
\	5C	•	•	•	
]	5D	•	•	•	
^	5E	•	•	•	
_	5F	•	•	•	
`	60	•			
a	61	•			
b	62	•			
c	63	•			
d	64	•			
e	65	•			
f	66	•			
g	67	•			
h	68	•			
i	69	•			
j	6A	•			
k	6B	•			
l	6C	•			
m	6D	•			
n	6E	•			
o	6F	•			
p	70	•			
q	71	•			
r	72	•			
s	73	•			
t	74	•			
u	75	•			
v	76	•			
w	77	•			
x	78	•			
y	79	•			
z	7A	•			
{	7B	•			
|	7C	•			

ISO/IEC 646 Character	Octet Value (HEX)	Included in Compaction Type			
		7-bit	6-bit	5-bit	Numeric
}	7D	•			
~	7E	•			

Annex E
(informative)

Encoding Example

The encoding example shown below is based on the processes specified in particular clauses in this international Standard. The relevant clause is referred to at each stage of the illustrated example.

E.1 Starting position

This stage is based on the process defined in 8.1.2

Table E.1 — Input Example of two ObjectIds and Associated Objects

Class Tag			Object		
Class Tag	**Length**	**ObjectId**	**Class Tag**	**Length**	**Object**
06	05	28 FC 59 0A 30	04	09	41 42 43 31 32 33 34 35 36
06	05	28 FC 59 0A 0D	04	02	35 30

E.2 The initial state of the entry for the Logical Memory

Using the example in Table E.1, the bytes can be mapped to create the initial encoding as illustrated in Table B.2. The Precursor bits are set to NULL.

All the other encoding is of bytes.

NOTE: The contents of Table E.2 will be updated as additional processes are applied.

Table E.2 — Initial Encoding example of Application Data

Precursor (bits)								Offset	L of ObjectId	ObjectId	L of Object	Object
b8	**b7**	**b6**	**b5**	**b4**	**b3**	**b2**	**b1**					
									05	28 FC 59 0A 30	09	41 42 43 31 32 33 34 35 36
									05	28 FC 59 0A 0D	02	35 30

E.3 The Logical Memory after data compaction

After processing through the Data Compactor, the data **object** and its length can be redefined; but the **objectId** remains unchanged.

The compaction process (as defined in 8.2 and Annex C) can be applied to the two data objects in Table E.1.

- The data object 41 42 43 31 32 33 34 35 36 is compacted to 7 bytes, using the 6 bit code Type, to become 04 20 F1 CB 3D 35 DA (see Annex C.4 for the detailed conversion).

- The data object 35 30 is compacted to 1 byte {32} using the integer Type (see Annex C.1 for the detailed conversion).

The first data object has the following changes:

- Compaction Type in the Precursor bits 7 to 5 = 100_{BIN}
- Length of data object = 07
- data object: 04 20 F1 CB 3D 35 DA

The second data object has the following changes:

- Compaction Type in the Precursor bits 7 to 5 = 001_{BIN}
- Length of data object = 01
- data object: 32

Table E.3 — Encoding Example After Data Compaction

Precursor (bits)								Offset	L of ObjectId	ObjectId	L of Object	Object
b8	b7	b6	b5	b4	b3	b2	b1					
	1	0	0						05	28 FC 59 0A 30	07	04 20 F1 CB 3D 35 DA
	0	0	1						05	28 FC 59 0A 0D	01	32

E.4 The Logical Memory after formatting with a noDirectory accessMethod

The format rules have a different effect depending on the **accessMethod** and **dataFormat** values in the system information. The following sub-clauses illustrate this using the same input with a **noDirectory accessMethod,** with different **dataFormats**.

E.4.1 The Logical Memory after formatting accessMethod = noDirectory, dataFormat = rootOidEncoded

Table E.4 shows the result of formatting the input of Table E.3 with **accessMethod = noDirectory** and **dataFormat = rootOidEncoded:**

- The root-OID 28 FC 59 0A is created, and its length of 4 bytes encoded in the Precursor.

- The first RELATIVE-OID is created. As it is a single arc it is a potential candidate for encoding in the Precursor, but as its value is greater than OE, the RELATIVE-OID has to be directly encoded in subsequent bytes and the Precursor bits 4 to 1 = 1111_{BIN}

- The Type of **objectId** and its length are encoded, as follows:

  - ► The Type is encoded in the first 3 bits; RELATIVE-OID = 100

  - ► The length is less than 16 bytes, so is encoded in bits 5 to 1 with Length = bbbbb-1; Length of 1 is encoded as 00010

  - ► The bit steam 10000010 = 82_{HEX}

- The second RELATIVE-OID is created. As it is a single arc it is a potential candidate for encoding in the Precursor, and as its value is no greater than OE, the RELATIVE-OID is encoded in the Precursor. RELATIVE-OID = 0D encoded in Precursor bits 4 to 1 = 1101_{BIN}

- No length nor value needs to be encoded for this RELATIVE-OID, because it is bit encoded in the Precursor, saving two bytes.

Table E.4 — Encoding Example of rootOidEncoded

Precursor (bits)								Offset	ObjectId Type & Length	ObjectId	L of Object	Object
b8	b7	b6	b5	b4	b3	b2	b1					
	0	0	0	0	1	0	0			28 FC 59 0A		
	1	0	0	1	1	1	1		82	30	07	04 20 F1 CB 3D 35 DA
	0	0	1	1	1	0	1				01	32

E.4.2 The Logical Memory after formatting accessMethod = noDirectory, dataFormat = di

Table E.5 shows the result of formatting the input of Table E.3 with **accessMethod = noDirectory** and **dataFormat = di.** This clearly shows that the root-OID is not encoded, because it is directly indicated by the particular **dataFormat.** In all other respects, the encodation is as in Table E.4, because the same procedures are used to create the precursor for each **objectId** / **object** pair.

Table E.5 — Encoding Example of dataFormat = di

Precursor (bits)								Offset	ObjectId Type & Length	ObjectId	L of Object	Object
b8	b7	b6	b5	b4	b3	b2	b1					
	1	0	0	1	1	1	1		82	30	07	04 20 F1 CB 3D 35 DA
	0	0	1	1	1	0	1				01	32

Annex F
(informative)

Logical Memory Structures

Various structures of Logical Memory are possible, depending on different factors including: the use of a directory and the encodation of particular root-OID values. This annex provides schematic layouts of the various structures.

F.1 Notation

The structures defined below use a simplified name for each component. The more comprehensive definition is as follows:

address =	Byte address in the directory of first byte of the corresponding data set (Precursor, lengthOID, **objectId,** length, **object)**
length =	Length of compacted data **object**
lengthOID =	Length of **objectId**
object =	compacted data **object** (null if Length = 0)
objectId =	root-OID, RELATIVE-OID, or OBJECT IDENTIFIER incorporates length of these Null if root-relative OID = 1 – 14
Precursor =	Precursor
Precursor(root-OID) =	Precursor of the root-OID when this has to be explicitly encoded
Terminator =	The Precursor set to 00_{HEX} acts as the terminator of the data sets at the beginning of the next highest (unused) block. It also acts as the terminator of the directory at the beginning of the next lowest (unused) block.
storageFormat	-- m,n where m = accessMethod (0-3) and n = dataFormat (0-31)

F.2 Non-directory structured Logical Memory with root-OID implicitly encoded

Where the root-OID is defined by the **dataFormat**, the following sequence applies:

storageFormat = 0,n (n = 1,3,4,5,8,9,10,11,12)

Precursor, lengthOID, **objectId,** length, **object** -- object 1
Precursor, lengthOID, **objectId,** length, **object** -- object 2
Precursor, lengthOID, **objectId,** length, **object** -- object 3
... -- continuation
Precursor, lengthOID, **objectId,** length, **object** -- object n
Terminator -- Precursor = 0 acts as terminator

NOTE: **storageFormat** = 0,1 requires the full OBJECT IDENTIFIER to be encoded. Although the root-OID is not implicit, the structure of the Logical Memory is as above.

F.3 Non-directory structured Logical Memory with root-OID explicitly encoded

Where the root-OID has to be directly encoded, the following sequence applies:

storageFormat = 0,2

Precursor(root-OID), **objectId,** 0 -- root-OID with zero length data
Precursor, lengthOID, **objectId,** length, **object** -- object 1
Precursor, lengthOID, **objectId,** length, **object** -- object 2
Precursor, lengthOID, **objectId,** length, **object** -- object 3
... -- continuation
Precursor, lengthOID, **objectId,** length, **object** -- object n
Terminator -- Precursor = 0 acts as terminator

F.4 Directory structured Logical Memory with root-OID implicitly encoded

Where the root-OID has to be directly encoded, the following sequence applies:

storageFormat = 1,n (n = 1,3,4,5,8,9,10,11,12)

Precursor, lengthOID, **objectId,** length, **object** -- object 1
Precursor, lengthOID, **objectId,** length, **object** -- object 2
Precursor, lengthOID, **objectId,** length, **object** -- object 3
... -- continuation of data sets
Precursor, lengthOID, **objectId,** length, **object** -- object n
Terminator (of data sets) -- Precursor = 0 acts as terminator
… -- unused tag memory
… -- unused tag memory
Terminator (of directory) -- Precursor = 0 acts as terminator
Precursor **objectId** address -- object n directory entry
... -- continuation of directory
Precursor **objectId** address -- object 3 directory entry
Precursor **objectId** address -- object 2 directory entry
Precursor **objectId** address -- object 1 directory entry

NOTE 1: **storageFormat** = 0,1 requires the full OBJECT IDENTIFIER to be encoded. Although the root-OID is not implicit, the structure of the Logical Memory is as above.

NOTE 2: Directory entries are stored in the lowest byte address to the highest byte address sequence within a block, but in reverse block sequence starting from the last block. So the Terminator logically follows directly after the last address byte of object n in the structure above.

F.5 Directory structured Logical Memory with root-OID explicitly encoded

Where the root-OID has to be directly encoded, the following sequence applies:

storageFormat = 1,2

Precursor(root-OID), **objectId,** 0 -- root OID with zero length data (must be at top to protect locked
-- **objectId** definitions)
Precursor, lengthOID, **objectId,** length, **object** -- object 1
Precursor, lengthOID, **objectId,** length, **object** -- object 2
Precursor, lengthOID, **objectId,** length, **object** -- object 3
... -- continuation of data sets
Precursor, lengthOID, **objectId,** length, **object** -- object n
Terminator (of data sets) -- Precursor = 0 acts as terminator
… -- unused tag memory

```
…                                   -- unused tag memory
Terminator (of directory)           -- Precursor  = 0 acts as terminator
Precursor objectId address          -- object n directory entry
...                                 -- continuation of directory
Precursor, objectId, address        -- object 3 directory entry
Precursor, objectId, address        -- object 2 directory entry
Precursor, objectId, address        -- object 1 directory entry
Precursor(root-OID), objectId, 0    -- root-OID with zero length data (repeated to reduce directory read
                                    -- accesses)
```

NOTE: Directory entries are stored in the lowest byte address to the highest byte address sequence within a block, but in reverse block sequence starting from the last block. So the Terminator logically follows directly after the last address byte of object n in the structure above.

ICS 35.040

Price based on 64 pages