INTERNATIONAL STANDARD

ISO/IEC 15962

Second edition
2013-03-15

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'objects — Protocole de données: règles d'encodage des données et fonctions logiques de mémoire

Reference number
ISO/IEC 15962:2013(E)

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*.

This second edition cancels and replaces the first edition (ISO/IEC 15962:2004), which has been technically revised.

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 RFID tag is invisible and accessible between the RFID tag and the interrogator only by the use of an air interface protocol, as specified in the appropriate part of ISO/IEC 18000. The result of the transfer of data between an application and an interrogator in open systems requires data to be encoded in a consistent manner on any RFID tag that is part of that open system. This is not only to allow equipment to be interoperable, but in the special case of data carriers, for the data to be encoded on the RFID 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 RFID tag must be formatted in such a way as to be reliably read at the point of use if the RFID tag is to fulfil its basic objective. This reliability is achieved through the specification of a data protocol using the application-defined arguments defined in ISO/IEC 15961-1 and the data encoding rules of this International Standard. Additionally, ISO/IEC 24791-1 specifies a software system infrastructure architecture that enables RFID system operations between business applications and RFID interrogators. Specific parts of ISO/IEC 24791 address data management requirements (ISO/IEC 24791-2) and device interface requirements (ISO/IEC 24791-5). These support defined implementations that incorporate the encoding rules of this International Standard and the functional rules of the commands and responses in ISO/IEC 15961-1.

Manufacturers of RFID equipment (interrogators, RFID tags, etc.) and the users of RFID technology require a standards-based data protocol for RFID for item management. ISO/IEC 15961-1 to ISO/IEC 15961-3, this International Standard, and ISO/IEC 24791 specify this protocol, which is layered above the air interface standards defined in ISO/IEC 18000.

The transfer of data to and from an application, supported by appropriate application commands, is the subject of ISO/IEC 15961-1. This International Standard specifies the overall process and the methodologies developed to format the application data into a structure to store on the RFID 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 ISO/IEC 15961. The encodation is in a Logical Memory as a software analogue of the physical memory of the RFID 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;

— specifies processes associated with sensory information and the transfers to the Tag Driver;

— defines other communication to the application.

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 15961-1, *Information technology — Radio frequency identification (RFID) for item management — Data protocol — Part 1: Application interface*

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

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

3 Terms, definitions and conventions

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
data compaction
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

3.1.2
Data Processor
implementation of the processes defined in this International Standard, including the Data Compactor, Formatter, Logical Memory, and Command/Response Unit

3.1.3
Precursor
byte, sometimes a sequence of bytes, used in the Directory and No-Directory Access-Methods that acts as metadata for the subsequent Object-Identifier and Object

3.1.4
Relative-OID
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

3.2 Conventions

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.

4 Conformance

Conformance to this International Standard shall depend on the functional capability of the device as defined in the following three sub-clauses.

4.1 Conformance with the air interface

A conformant implementation of this International Standard shall support one or more air interface protocols through the tag drivers defined in Annex C. Declarations of conformance shall refer to the specific air interface protocol(s). This applies to encoders, decoders, or more comprehensive devices.

4.2 Conformance with the application interface

The conformance requirements depend on the type of device as follows:

4.2.1 Encoders and the application interface

Within the constraints of the air interface protocol supported, a conformant implementation of this International Standard on an encoder shall support the application commands defined in 8.2 and the associated process argument, as defined in Clause 10.

A conformant RFID tag shall have its encoding in a state that can be properly decoded by a conformant decoder (see 4.2.2).

4.2.2 Decoders and the application interface

Within the constraints of the air interface protocol supported, a conformant implementation of this International Standard on a decoder shall support the application commands defined in 8.3 and the associated process argument, as defined in Clause 10.

4.2.3 Comprehensive encoder/decoder devices and the application interface

Within the constraints of the air interface protocol supported, a conformant implementation of this International Standard on an encoder/decoder shall support the application commands defined in 8.2 and 8.3 and the associated process argument, as defined in Clause 10. In addition, the **Delete-Object** (see 8.4.2) and **Modify-Object** (see 8.4.3) commands shall be supported. Other commands defined in 8.4 may be supported, and each command that is supported shall be declared.

4.3 Conformance with the Access-Method

The conformance requirements depend on the type of implementation as follows:

4.3.1 Encoders and the Access-Method

A conformant implementation of this International Standard on an encoder shall support the encoding rules and formatting rules of one or more **Access-Methods** as defined in Clause 11 and associated Annexes. Declarations of conformance shall refer to the specific **Access-Method(s**) supported.

4.3.2 Decoders and the Access-Method

A conformant implementation of this International Standard on a decoder shall support the decoding rules and formatting rules of all the **Access-Methods** as defined in Clause 11 and associated Annexes.

An interrogator is not expected to fully support the decoding functions of all the Access-Methods, and the following shall apply to achieve conformance:

- For full conformance, the decoder process on the interrogator shall output the **Object-Identifier**, **Object** and other arguments as required in the responses to the commands.
- For partial conformance, the decoder process on the interrogator shall output the byte string that represents the encoded package (depending on the **Access-Method**) containing the requested **Object-Identifier**. The encoded package then needs to be fully decoded by a decoder process, external to the interrogator, that is fully compliant with the rules defined in this International Standard.

Declarations of conformance shall refer to the specific **Access-Method(s**) supported.

4.3.3 Comprehensive encoder/decoder devices and the Access-Method

A conformant implementation of this International Standard on an encoder/decoder shall support the encoding rules and formatting rules of one or more **Access-Methods** as defined in Clause 11 and associated Annexes. Declarations of conformance shall refer to the specific **Access-Method(s**) supported. The decoding function shall be as defined in 4.3.2.

5 Protocol model

5.1 Overview

RFID supports bit encodation in the RFID 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 RFID tag memory. However, as an RFID 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 15961-1 is to specify a common interface between the application programs and the RFID interrogator. The prime function of this International Standard is to specify the common encoding rules and logical memory functions.

RFID tags utilise electronic memory, which is typically capable of increasing data capacity as new generations of product are introduced. Differences in data capacity of each RFID 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.

5.2 Layered protocol

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

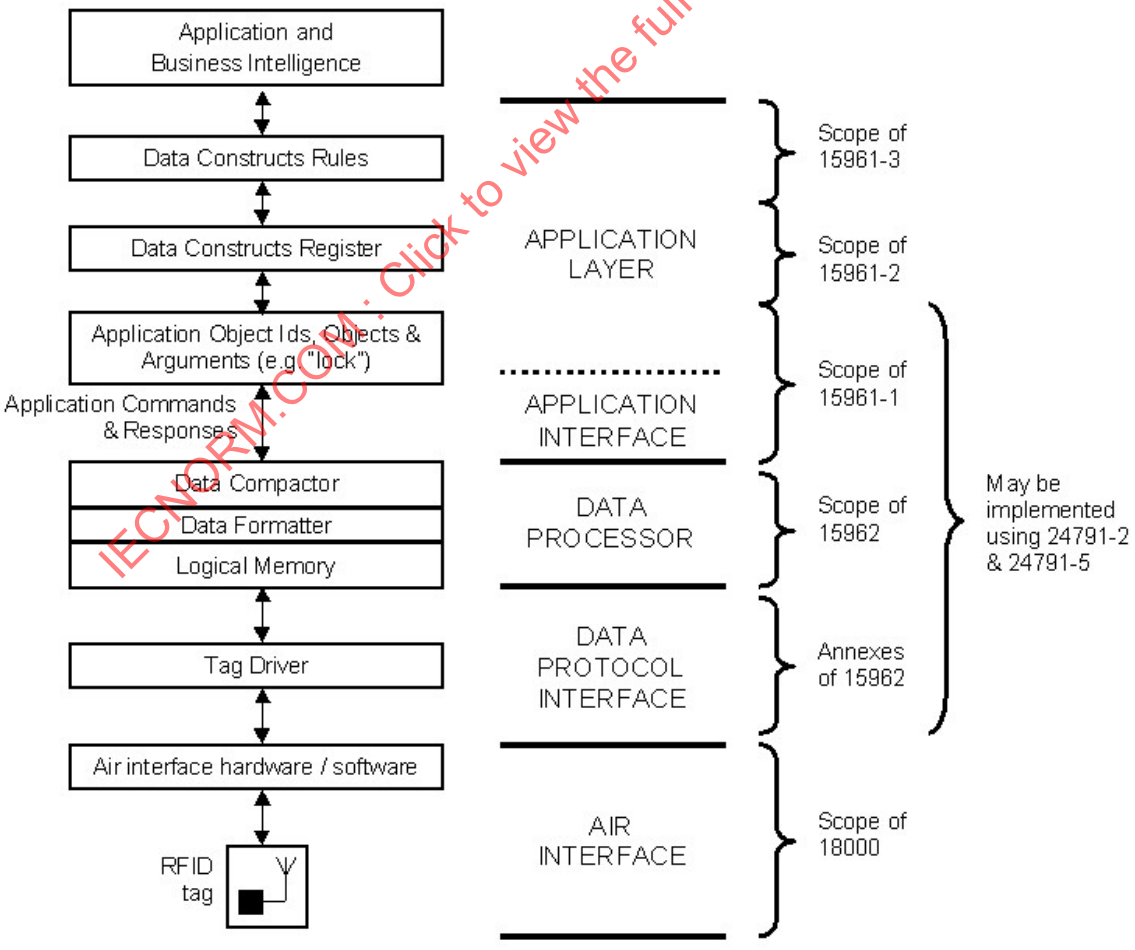


Figure 1 — Schematic of protocol layers for an implementation of RFID for item management

5.2.1 Application layer - as defined in the various parts of 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 RFID tag. ISO/IEC 15961-3 specifies the data construct rules for the AFI, DSFID, object identifier for the unique item identifier, and object identifier structure for other item-related data. This ensures that each piece of data can be uniquely identified, both within the scope of a particular application and between applications.

Each application needs to be registered according to the rules of ISO/IEC 15961-2 so that the data constructs can be declared and used in an unambiguous manner.

The RFID data protocol in ISO/IEC 15961-1 defines functions and arguments used to construct application commands and responses. This is so that application programs can specify what data to transfer to and from the RFID tag and to append, update, selectively lock, delete data, or perform other functions on the RFID tag.

To illustrate how the functions and arguments are assembled into a structured format, a number of commands and responses have been constructed using an abstract syntax. This 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. The abstract syntax used in ISO/IEC 15961-1 is similar to that used in ISO/IEC 24791-5, and is intended to enable closer integration with that standard. The original version of ISO/IEC 15961:2004 included commands defined using ASN.1 abstract syntax. For backward compatibility the commands that were originally defined in this manner have been included in an annex of ISO/IEC 15961-1.

This RFID data protocol also defines arguments and codes to support responses of data that is read from an RFID tag, including error messages, which are returned to the application.

The abstract syntax may be used as a basis to prepare commands in different programme languages, supporting the functionality and arguments of the abstract commands.

5.2.2 Application interface - as defined in ISO/IEC 15961-1

The application interface may be implemented in a number of different ways that are not explicitly defined in this International Standard, nor in ISO/IEC 15961-1. The basic requirement is to identify data objects distinctly from all others using object identifiers, even to enable different data formats to be intermixed on the same RFID tag. The application interface also needs to define command and response arguments unambiguously, so that they can be intermixed with data on the same wired or wireless network.

One major class of implementation, described as a *straight-through process*, is appropriate where the functions and arguments used to construct commands and the arguments and codes used to construct responses, as specified in this International Standard, are directly input to the encoding processes of ISO/IEC 15962. Such input can be from computer screens or forms, or more direct transfers from host systems. The advantage of this process is that it avoids the creation of the transfer encoding (see below), but requires more rigorous adherence to the functional requirements of the commands and responses. ISO/IEC 15961-1 imposes no constraints on the particular application interface process to be adopted, other than the requirement that it be integrated with the encoding rules of ISO/IEC 15962.

An alternative process, consistent with the first edition of ISO/IEC 15961, is to use the abstract syntax for defining the commands and responses in a structured, consistent and verifiable manner. It is then necessary to generate the transfer encoding that defines the byte stream transferred between the processes of this International Standard and those of ISO/IEC 15962.

Whichever approach is used, the encoding rules of ISO/IEC 15962 shall be followed, and the encoding on the RFID tag has to be complaint with all the arguments in the commands specified in ISO/IEC 15961-1.

5.2.3 Data Protocol Processing - as defined in this International Standard

The RFID data protocol specifies how data is encoded, compacted and formatted on the RFID tag and how this data is retrieved from the RFID tag to be meaningful to the application.

This RFID data protocol provides for a set of schemes that compact the data to make more efficient use of the memory space.

This RFID data protocol also supports various storage formats to enable efficient use of memory and efficient access procedures.

5.2.4 Data Protocol Interface - as defined in this International Standard

Each air interface protocol standard in ISO/IEC 18000 has its own specific rules for defining commands and responses. Furthermore, some air interface protocols can support different tag architectures with different memory sizes, and possibly support optional commands. The data protocol provides a mechanism to interface with these rules through specific tag drivers. These allow the basic application commands and responses of ISO/IEC 15961-1 to be applied independently of the air interface protocol and specific tag architecture.

The tag driver component of the data protocol provides the mapping rule from the generic processes to the specific tag requirements. These mapping rules are used to write data and to read data.

Additional tag drivers can be specified as new air interface protocols are introduced in the ISO/IEC 18000 series of standards.

5.3 Flexible implementation configurations

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

— with the International Standard incorporated into the software system infrastructure architecture defined in ISO/IEC 24791. This is the recommended approach for any networked application.

— with ISO/IEC 15961 Part 1 and this International Standard incorporated into stand-alone software or devices that have as its output conformant encoding and / or conformant decoding with responses compliant to the responses of ISO/IEC 15961-1.

5.4 Functional processes – interrogator implementation

There are various functional processes that need to take place to write data to an RFID tag and to read data from it. Figure 2 — Logical functions and interfaces of ISO/IEC 15962 with other RFID system components shows a schematic of an example 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.

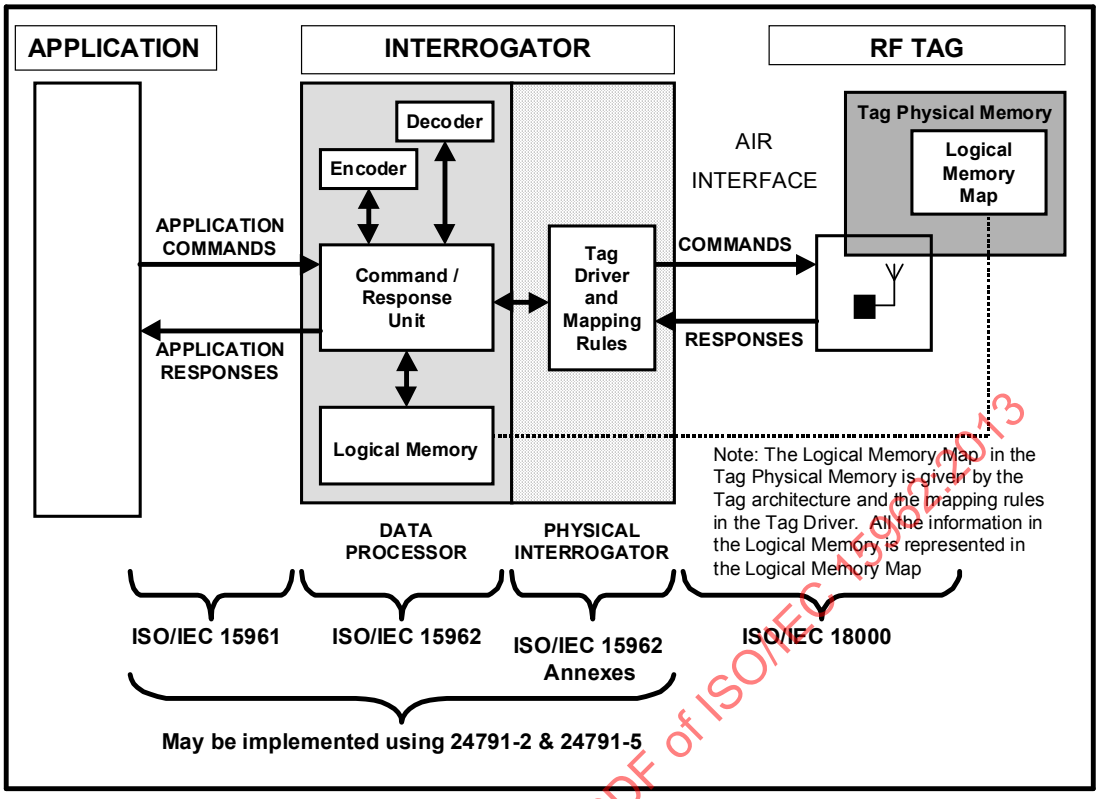


Figure 2 — Logical functions and interfaces of ISO/IEC 15962 with other RFID system components

5.4.1 Functional processes – application interface

The data flows between the application and the Data Processor are formatted according to this International Standard and are not compacted. 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 this International Standard and ISO/IEC 15961-1. 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.

5.4.2 Functional processes – interrogator

In the process illustrated in Figure 2 — Logical functions and interfaces of ISO/IEC 15962 with other RFID system components, the interrogator is the module in which all the basic processing of the data protocol takes place and there is an interface to the RFID tag. Different implementations might separate some of the functions described below and have an interface between the application and the physical interrogator.

5.4.2.1 Data processor

The Data 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 Processor can physically reside anywhere between the application software and the tag driver but shall contain all the components. Some, or all, of the Data Processor functions may be implemented using processes defined in ISO/IEC 24791-2 and ISO/IEC 24791-5.

5.4.2.1.1 Command/Response unit

The Command/Response Unit receives the application commands from the application in a format specified in this International Standard, acting upon these commands where appropriate and converting to the specific RFID 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 Processor. Information from the particular RFID tag is required to set the parameters of the Logical Memory Map (e.g. number of bytes, whether a directory is in use, etc) on the RFID 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 logical boundary between the Data Processor and the Tag Driver.

5.4.2.1.2 Logical Memory

This is an array of contiguous bytes of memory acting as a common software representation of the RFID tag memory accessible by an application and to which the object identifiers and data objects are mapped in bytes. The Logical Memory takes into account some parameters of the real RFID 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 RFID tag, but that individual RFID 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 RFID tags, without changing the application.

5.4.2.1.3 Encoder

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

5.4.2.1.4 Data Compactor

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

5.4.2.1.5 Formatter

The 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 RFID tag architecture is performed based on information provided by the Tag Driver.

5.4.2.1.6 Decoder

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

5.4.2.2 Tag Driver

The Tag Driver provides two main functions:

— It provides mapping rules on the structure of the RFID tag to enable the contents of the Logical Memory in the Data Processor to be exchanged with the Logical Memory Map of the RFID 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 RFID Tag. For example, an application command **write Data Object {name}** could result in the RFID tag related command of write (block #, data).

The description of the tag driver for particular RFID 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 RFID 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.

5.4.2.3 Transfer mechanisms

The transfer mechanisms for transferring data, commands and responses between the Data Processor (i.e. the implementation of this International Standard) and the RFID tag need to be done through the interrogator or incorporated within the functions of the interrogator.

5.4.3 RFID tag

Although the RFID tag is beyond the scope of this International Standard, the tag is shown in Figure 2 — Logical functions and interfaces of ISO/IEC 15962 with other RFID system components to complete the flow of data and commands. Within the RFID tag, the Logical Memory Map represents all the data in the Logical Memory of the Data Processor converted (or mapped) to a location structure determined by the mapping rules in the Tag Driver and the architecture of the RFID tag.

5.5 ISO/IEC 15962 and the Data Processor

This International Standard defines all the rules for encoding data on an RFID tag. The implementation of these rules is described in this International Standard as the Data Processor. As described in 5.4, various processes are undertaken to achieve successful encoding. The rules ensure the encoded bytes are 'self-declaring' in a reading operation without any previous or independent knowledge of what is encoded on the tag. This allows the Data Protocol to be used in open systems where the organisation encoding the data on the RFID tag might be completely unaware of what organisation might eventually read the data from the RFID tag. This is achieved by:

— Having clearly defined rules for compacting data that are applied independently of the application and the type of RFID tag.

— Defining fundamental structuring rules for the encoding of data largely determined by a user-selectable feature called the **Access-Method**.

— Having precise structuring rules within each **Access-Method**, including a clearly defined syntax that allows reading systems to selectively identify data required for the application.

Clause 11 provides a description and a definition of the **Access-Methods** currently supported. The associated processing is fully specified in Annexes referred to in Clause 11.

6 Data and presentation conventions

The following conventions are used:

— The field/argument names are shown in **bold** type face using this format: **Access-Method**.

— Basic data types are shown in UPPERCASE type face using this format: BOOLEAN

6.1 Data types in ISO/IEC 15961-1 commands and responses

The following data types are used in the commands and responses:

— BOOLEAN: An argument that can have the values TRUE or FALSE.

— BIT STRING: A sequence of bits.

— BYTE: An integer with the possible values 0 to 255, usually expressed as a hexadecimal value 00h to FFh.

— BYTE STRING: A sequence of bytes. (Equivalent to OCTET STRING)

— HEXADECIMAL ADDRESS: A location (on the memory of an RFID tag), expressed as a hexadecimal value

— INTEGER: An integer can take any whole number. In the context of this International Standard, the values are all positive.

— OBJECT IDENTIFIER: An Object Identifier as defined in 6.3.1.

6.2 Extensible bit vector (EBV)

The EBV block structure is used as a self-declaring code to determine the length, position or size of some characteristic. Various EBV-n structures are used in this International Standard, where n refers to the number of bits in each block. EBV-8 is identical to the length encoding (see 9.2.10), which has been used since the original edition of this International Standard.

6.3 Object Identifier presentation in the application interface

6.3.1 Object identifier structure to ISO/IEC 8824-1

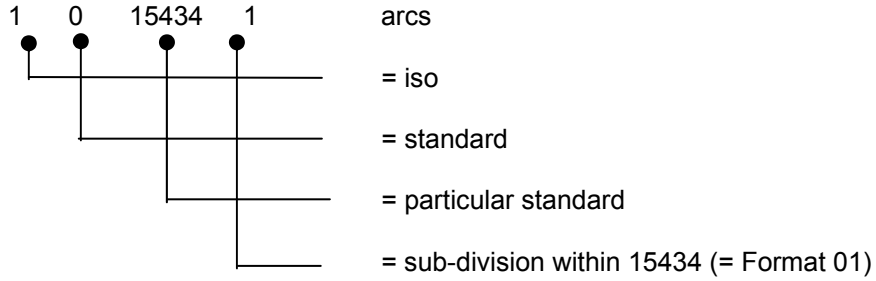
This International Standard uses the OBJECT IDENTIFIER type as defined in ISO/IEC 8824-1 and with identifiers assigned as specified in ISO/IEC 9834-1. This uses a registration tree with a common implied root node (ISO/IEC 9834-1), a series of arcs from each node, with new arcs added as required to define a particular object. Thus, the body responsible for a particular node:

— has a defined set of arcs to identify itself

— can manage the allocation of arcs under its node, independently of other bodies

— is assured of uniqueness from all other arcs in the registration tree

EXAMPLE



The only top arcs permitted for all object identifiers are shown in Table 1 — Object Identifier Top Arcs.

Table 1 — Object Identifier Top Arcs

Identifier Arc Name	Numeric Value
itu-t	0
iso	1
joint-iso-itu-t	2

The second arc is administered by the relevant organisation named for the top arc. The current list of top and second arcs is given in ISO/IEC 15961 Part 3.

The third arc is controlled by the system or body defined for the second arc; sometimes this is a Registration Authority. The hierarchical structure continues until the object is identified uniquely. The procedure of naming object identifiers ensures that each object is unique within its "parent" arc and that each parent arc is unique within its previous level, right back to the top 3 arcs.

NOTE This structure enables object identifiers from different domains (e.g. open and closed systems) to be encoded unambiguously on an RFID tag memory.

Three forms of object identifier are used with the RFID Data Protocol:

— **Object-Identifier**: This full structure is used for communications between the application and the Data Processor defined by the scope of this International Standard. The full structure is suitable for use in this International Standard where the set of object identifiers to be encoded on the RFID tag have different higher level arcs, or where a generic system is in place.

— **Relative-OID**: This structure is used in conjunction with the **Root-OID** (see below) for communication between the application and the Data Processor. These structures are applied in situations where a common root applies to the set of object identifiers to be encoded on the RFID tag. For example, if all the object identifiers have the common root 1 0 15961 12 encoding space can be saved on the RFID tag if this common **Root-OID** does not have to be encoded for each object identifier. The **Relative-OID** is a suffix to a common **Root-OID**, which is either encoded or declared in some other way.

— **Root-OID**: The **Root-OID** is the common part of a set of encoded object identifiers. It acts as a common prefix to the **Relative-OID** values encoded on the RFID tag. This structure is particularly important in applications that require a variety of data from a common data dictionary to be encoded on an RFID tag. The **Root-OID** is either explicitly encoded or declared in some other way, according to the encoding rules of this International Standard.

6.3.2 Presenting the Object-Identifier in the style of ISO/IEC 8824-1

When the **Object-Identifier** is presented in application commands and responses in the style of ISO/IEC 8824-1, spaces are inserted between each arc as follows:

1 0 15961 12 1

6.3.3 Presenting the Object-Identifier as a Uniform Resource Name (URN)

The **Object-Identifier** may also be presented in the URN in the following format, based on IETF RFC 3061, with the decimal point character between each arc:

urn:oid:1.0.15961.12.1

6.4 The Object

On the interface with the application, all data that is intended to be encoded on the RFID is described as the **Object** in this International Standard. The **Object** shall be qualified by the use of an **Object-Identifier** to ensure unambiguous communication.

When the **Object-Identifier** is that of a unique item identifier then its combination with the **Object** confers domain uniqueness. Other **Object-Identifier** plus **Object** pairs (e.g. to identify a particular batch number) are not unique, but represent attribute data that is associated with the unique item identifier.

6.5 The 8-bit byte

This International Standard supports the encoding of data and data objects aligned as 8-bit bytes (sometimes referred to as octets 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 defined in Table 2 — Bit sequence in the 8-bit byte.

Table 2 — Bit sequence in the 8-bit byte

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

NOTE Although other bit position conventions are possible and more common, this bit sequence is used here for backward compatibility with the original version of this International Standard, which in turn complied with ASN.1 rules.

6.6 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.

7 Data Processor – high level processing

The main flows of information to and from the command/response unit are illustrated in Figure 3 — High level information flows, which focuses on the encoding process.

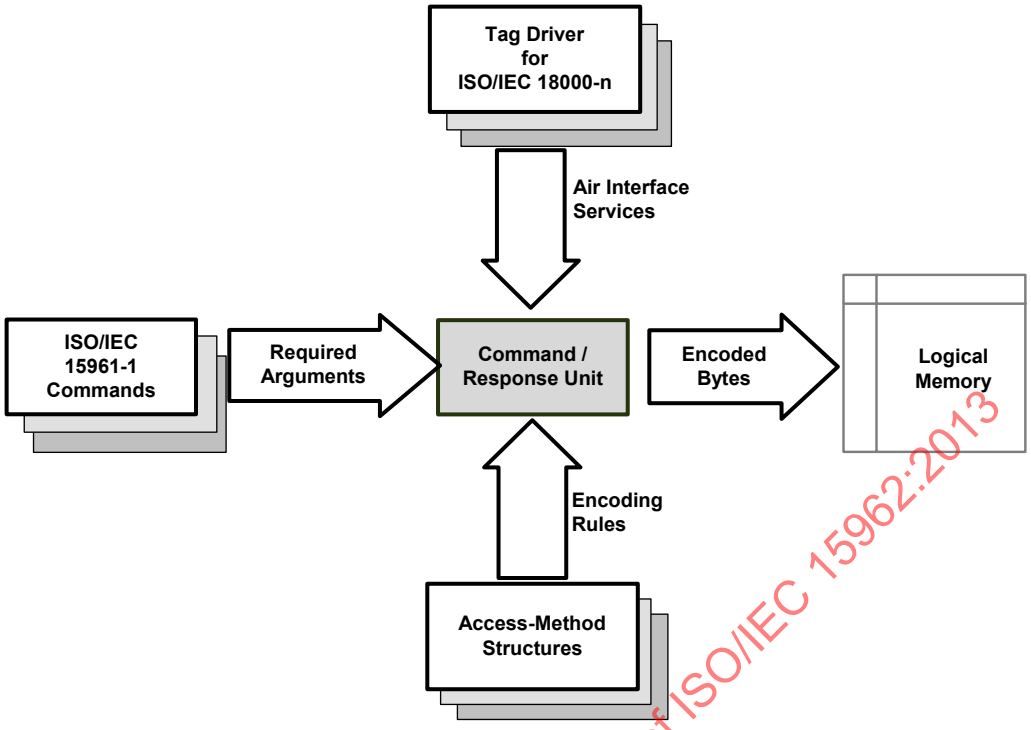


Figure 3 — High level information flows

The Data Processor is not required to support all application commands. For the commands that are supported, all the set of incorporated arguments shall be processed correctly to achieve the encoding required by the application. These requirements are defined in Clause 8. The method of transferring the commands to the command response unit is not specified in this International Standard.

Applications should define that the data is to be encoded on a particular type of RFID tag. The Tag Driver for the particular tag provides the air interface services as defined in Clause 9. These are used by the Command / Response Unit to correctly encode data on, and decode data from, the type of tag required by the application. The Data Processor is not required to support all tag types, but shall support the encoding and decoding capabilities of any type of tag that is supported. As some types of RFID tag that are defined in the ISO/IEC 18000 series of air interface protocols have optional components, the precise capability of the Data Processor needs to be clearly defined against optional components and Revisions to the ISO/IEC 18000 series of standards.

8 Data Processor and the application interface

8.1 Application commands – overview

ISO/IEC 15961-1 defines a number of commands and associated responses to provide instructions of how data is to be encoded on the RFID tag. Table 3 — ISO/IEC 15961-1 commands lists these commands. The column headed "Code" identifies the final arc of the object identifier that identifies the command module and response module. These code values have been retained to provide a structure that is compatible with the original publication (ISO/IEC 15961:2004).

EXAMPLE The inventory-tag command has the object identifier: 1 0 15961 126 3. Its response has the object identifier: 1 0 15961 127 3.

Table 3 — ISO/IEC 15961-1 commands

Code	Command Name	Write	Read	Other
1	Configure-AFI	Note 1		
2	Configure-DSFID	Note 1		
3	Inventory-Tags			YES
5	Delete-Object			YES
6	Modify-Object			YES
8	ReadObject-Identifiers		YES	
10	Read-Logical-Memory-Map		YES	
12	Erase-Memory			Note 2
13	Get-App-based-System-Info			Note 3
17	Write-Objects	YES		
18	Read-Objects		YES	
19	Write-Objects-Segmented-Memory-Tag	YES		
20	Write EPC-UII	YES		
21	Inventory-ISO-UIImemory		YES	
22	Inventory-EPC-UIImemory		YES	
23	Write-Password-Segmented-Memory-Tag	YES		
24	Read-Words-Segmented-Memory-Tag		YES	
25	Kill-Segmented-Memory-Tag			YES
26	Delete-Packed-Object			YES
27	Modify-Packed-Object			YES
28	Write-Segments-6TypeD-Tag	YES		
29	Read-Segments-6TypeD-Tag		YES	
30	Write-Monomorphic-UII	YES		
31	Configure-Extended-DSFID	YES		
32	Configure-Multiple-Records-Header	YES		
33	Read-Multiple-Records		YES	
34	Delete-Multiple-Record			YES

NOTE 1. The Configure-AFI and Configure-DSFID commands are implemented in different ways, depending on the air interface protocol. Some support specific air interface commands that are distinctly different from the generic write commands (e.g. ISO/IEC 18000-3M1). Others use the write command, but require the data to be written into a specific location (e.g. ISO/IEC 18000-6B and ISO/IEC 18000-6C).

2. Some air interface protocols support a specific Erase-Memory command, whereas others require the data to be overwritten with a null-byte value (typically 00_{16}).

3. The Get-App-Based-System-Info command is one that is explicitly supported in some air interface protocols but in others is completely redundant, because the information is returned as part of another command.

Table 3 — ISO/IEC 15961-1 commands has three columns: write, read, other. The **write** column shows the commands that are most likely to be implemented in a device such as a printer-encoder, and in an application that simply provides source marking for data capture at a later stage and at a different point. The **read** column identifies commands that need to be implemented in the simplest of data capture devices. The commands highlighted in the **other** column either require more complex processing (e.g. Modify-Object) or require a special process (e.g. Kill-Segmented-Memory-Tag).

The commands associated with code values not included in Table 3 — ISO/IEC 15961-1 commands were originally specified in ISO/IEC 15961:2004. This International Standard no longer needs to support these commands, mainly because their functionality has been absorbed in other commands. The original commands may continue to be supported by encoding systems compliant with ISO/IEC 15962:2004.

To implement the encoding rules of this International Standard it is necessary to understand the detail of the commands and responses in ISO/IEC 15961-1. The sub-clauses that follow, provide a listing of commands under the three main headings of: read, write, and other. The arguments for each command are listed in the sub-clauses.

8.2 Application commands and responses– write

8.2.1 Configure-AFI

The **AFI** is a single byte code and is used as part of the selection process in an application. Depending on the air interface protocol, it may written into a particular location of memory using an explicit air interface protocol command. This application command is intended to support this process. It may also be used to support the encoding of the **AFI** using a generic air interface write command.

8.2.1.1 Process requirements

To correctly implement this command in the Data Processor, in addition to processing the **AFI** in the range 90_{16} to CF_{16}, the following process argument shall be supported:

AFI (see 10.1.3)
AFI-Lock (see 10.1.4)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
1 AFI-Not-Configured
2 AFI-Not-Configured-Locked
3 AFI-Configured-Lock-Failed
8 Singulation-Id-Not-Found
255 Execution-Error

8.2.1.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— That the RFID tag(s) supported by the encoder have the **AFI** in a memory location requiring an explicit air interface command, or that a generic air interface write command supports writing the AFI to a specific location determined by the tag driver with a single byte write transaction.

— If the encoder supports writing the AFI, then it shall also support the transfer of any **AFI-Lock** argument so that it can be implemented in the air interface.

— The encoder shall support the associated **Completion-Codes** and **Execution-Codes**.

8.2.2 Configure-DSFID

The **DSFID** is a single byte or multiple byte code that is used to reduce the encoding of **Object-Identifiers**, based on encoding rules in this International Standard. In particular, the **DSFID** specifies the **Access-Method** and the **Data-Format** assigned to particular applications. Depending on the air interface protocol, the **DSFID** is written into a particular location of memory using an explicit air interface protocol command. This application command is intended to support this process. It may also be used to support the encoding of the **DSFID** using a generic air interface write command.

8.2.2.1 Process requirements

To correctly implement this command in the Data Processor, in addition to processing the **DSFID**, the following process argument shall be supported:

DSFID (see 10.1.11)
DSFID-Lock (see 10.1.12)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
4 DSFID-Not-Configured
5 DSFID-Not-Configured-Locked
6 DSFID-Configured-Lock-Failed
8 Singulation-Id-Not-Found
255 Execution-Error

8.2.2.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— That the RFID tag(s) supported by the encoder have the **DSFID** in a memory location requiring an explicit air interface command, or that a generic air interface write command supports writing the **DSFID** to a specific location determined by the tag driver with a single byte write transaction.

— If the encoder supports writing the **DSFID**, then it shall also support the transfer of any **DSFID-Lock** argument so that it can be implemented in the air interface.

— The encoder shall support the associated **Completion-Codes** and **Execution-Codes**.

8.2.3 Write-Objects

This command shall not be used to write a **Monomorphic-UII**. If the **AFI** on the RFID tag declares that it is registered for a **Monomorphic-UII,** the appropriate error shall be returned and the encoding process aborted. The correct command to use is defined in 8.2.8.

The **Write-Objects** command is used to write one or more **Object-Identifiers** and associated **Objects** to an RFID tag. This command may be implemented to write the initial data to the RFID tag, or to add data to the tag. The command is supported by the following compound arguments, for which individual arguments are listed below:

Add-Objects-List
DSFID-Constructs
Ext-DSFID-Constructs
Multiple-Records-Constructs
Packed-Object-Constructs

8.2.3.1 Process requirements

To correctly implement this command in the Data Processor, the process arguments and compound arguments defined for the command in ISO/IEC 15961-1 shall be supported. For Multiple Records they are applied once per record, otherwise they are applied once per Logical Memory.

Data-CRC-Indicator (see 9.2.11)
DSFID (see 10.1.11)
DSFID-Lock (see 10.1.12)
DSFID-Pad-Bytes (see 9.2.17)
Length-Of-Encoded-Data (see 9.2.10)
Memory-Capacity (see 9.2.10)
Tag-Data-Profile-ID-Table (see 10.1.53)

The main determinant for compliant encoding is the **Access-Method**, which is incorporated as part of the **DSFID**. The **DSFID** is provided as an optional argument for use in one of the following ways:

— If data is being written to a blank RFID tag, then the **DSFID** is provided as part of this command to minimise communications.

— If data is being added to the RFID tag, then the **DSFID** in the command should match the **DSFID** already encoded on the RFID tag, else there is an error and the encoding process can cease before significant amounts of data have been processed.

To correctly implement this command in the Data Processor, the following process arguments shall be supported for each **Object-Identifier** and **Object pair**, irrespective of the **Access-Method**:

Avoid-Duplicate (see 10.1.7)
Compact-Parameter (see 10.1.9)
Object-Lock (see 10.1.38)

The **Avoid-Duplicate, Compact-Parameter** and **Object-Lock** arguments shall be applied in a manner consistent with the rules of the **Access-Method** (see Clause 11) to ensure reliable encoding. If there is an inconsistency within the application command, then it shall be considered to be in error and the encoding process shall not be implemented.

If the **Access-Method** = **Packed-Objects**, the input states are all defined in the **Packed-Objects-Constructs** argument specified in ISO/IEC 15961-1 and apply to a single Packed Object. That argument in turn has the following arguments, for which the processes are defined in Annex I:

Block-Align-Packed-Objects
Editable-Pointer-Size
ID-Type
Object-Offsets-Multiplier
Packed-Object-Directory-Type
PO-Directory-Size
PO-ID-Table
PO-Index-Length

If the **Access-Method** = **Tag-Data-Profiles**, the input states are all defined by the **Tag-Data-Profile-ID-Table** and encoded as a single Tag Data Profile using the processes defined in Annex N.

The following tag-related arguments, if declared by the Extended DSFID are used as follows to complete the encoding process:

— If the memory **Memory-Capacity** (see 9.2.10) is encoded, or the air interface has its own means of providing this information, it can be used to assess in there is sufficient memory for the command to be fully implemented.

— If one or both **Data-CRC-Indicators** (see 9.2.11) is set, then the CRC-16 shall be applied as defined in 9.2.12.

— If the **Length-Of-Encoded-Data** (see 9.2.10) is signalled to be encoded, then this shall be calculated after all encoded bytes have been determined. This field is then written, or overwritten, taking into account the availability of **DSFID-Pad-Bytes** (see 9.2.17).

If the compound argument **Multiple-Records-Constructs** is used in the command, then the **Object-Identifier** structure as define in 11.5 shall be used for identifying each data **Object** in the **add-Objects-List** argument. If this is not the case, then the command shall not be processed. In addition to writing the Multiple-Record itself, this command can require other areas of the Logical Memory to be encoded: The MR-header if the number of records is maintained and the directory if already encoded or called for by the command arguments. If any part of the memory required to implement the command is locked, then the command shall be aborted.

Encoding shall be as defined in Annex Q if the first four arcs are **1.0.15961.401**, or as defined in Annex R if the first four arcs are 1.0.15961.402 or 1.0.15961.403. The compound argument in turn has the following arguments, for which the processes are defined in Annex Q and Annex R.

Append-To-Existing-Multiple-Record (see 10.1.5)
Application-Defined-Record-Capacity (see 10.1.6)
Identifier-Of-My-Parent (see 10.1.17)
Lock-Directory-Entry (see 10.1.24)
Lock-Record-Preamble (see 10.1.26)
Number-In-Data-Element-List (see 10.1.36)
Record-Memory-Capacity (see 10.1.46)
Record-Type-Classification (see 10.1.48)
Update-Multiple-Records-Directory (see 10.1.58)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)
Object-Identifier (see 6.3)

The response arguments **Completion-Code** and **Object-Identifier** are associated with each data element.

The **Completion-Codes** for this command will vary based on the associated compound arguments. The complete list is:

0 No-Error
8 Singulation-Id-Not-Found
9 Object-Not-Added
10 Duplicate-Object
11 Object-Added-But-Not-Locked
29 Object-Not-Editable
31 Packed-Object-ID-Table-Not-Recognised-No-Encoding
32 Tag-Data-Profile-ID-Table-Not-Recognised
33 Insufficient-Tag-Memory
36 Command-Cannot-Process-Monomorphic-UII
37 Data-CRC-Not-Applied
38 Length-Not-Encoded-In-DSFID 255 Execution-Error
43 Data-Format-Not-Compatible-Multiple-Records-Header
44 Access-Method-Not-Compatible-Multiple-Records-Header
45 Sector-Identifier-Not-Compatible-Multiple-Records-Header
46 Record-Preamble-Not-Configured
47 Record-Preamble-Not-Locked
255 Execution-Error

8.2.3.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— The encoder supports all aspects of the **Access-Method**.

— The encoder supports all the processed defined by the Extended DSFID

— The encoder supports the associated **Completion-Codes** and **Execution-Codes**.

— If any of the first two conditions are not supported an error is signalled by returning the **Execution-Code** = 4 **Command-Not-Supported**.

8.2.3.3 Guidance for appending data

To maximize reliability when adding additional data or modifying data, it is recommended that the interrogator performs the following procedure:

a) write all the new bytes except the first word or block of data (i.e., the data that will overwrite the existing 00_{16} terminator), including the new terminator at the end of the new data;

b) optionally read all of the new encoded data to ensure that it was stored correctly, then

c) write the first word/block, which overwrites the original terminator.

NOTE Writing the new data in this sequence, preserves the ability to parse the pre-existing data in an event of a failure to write of the new data.

8.2.4 Write-Objects-Segmented-Memory-Tag

This command shall not be used to write a **Monomorphic-UII**. If the **AFI** on the RFID tag declares that it is registered for a **Monomorphic-UII,** the appropriate error shall be returned and the encoding process aborted. The correct command to use is defined in 8.2.8.

The **Write-Objects-Segmented-Memory-Tag** command is similar to the **Write-Objects** command except that it is intended to write data to a selected memory bank in a segmented memory tag. The command may be implemented to write initial data to the RFID tag, or to add data to the tag. The command is supported by the following compound arguments , for which individual arguments are listed below:

Add-Objects-List
DSFID-Constructs
Ext-DSFID-Constructs
Multiple-Records-Constructs
Packed-Object-Constructs

8.2.4.1 Process requirements

To correctly implement this command in the Data Processor, the process arguments and compound arguments defined for the command in ISO/IEC 15961-1 shall be supported. For Multiple Records they are applied once per record, otherwise they are applied once per Logical Memory.

Access-Password (see 10.1.1)
AFI (see 10.1.3)
Data-CRC-Indicator (see9.2.11)
DSFID (see 10.1.11)
DSFID-Lock (see 10.1.12)
DSFID-Pad-Bytes (see 9.2.17)
Length-Of-Encoded-Data (see 9.2.10)
Memory-Bank (see 10.1.29)
Memory-Capacity (see 9.2.10)
Tag-Data-Profile-ID-Table (see 10.1.53)

The **Access-Password** in the application command shall match that encoded on the RFID tag to continue processing. If there is a mismatch, then an error shall be reported.

The main determinant for compliant encoding is the **Access-Method**, which is incorporated as part of the **DSFID**. The **AFI** and **DSFID** are provided as arguments for use in one of the following ways:

— If data is being written to a blank segmented memory RFID tag, then these arguments are provided as part of this command to minimise communications.

— Depending on the memory bank concerned, if data is being added to the segmented memory RFID tag, then the **AFI** and **DSFID** in the command should match the **AFI** and **DSFID** already encoded on the RFID tag, else there is an error and the encoding process can cease before significant amounts of data have been processed.

To correctly implement this command in the Data Processor, the following process arguments shall be supported for each **Object-Identifier** and **Object pair**, irrespective of the **Access-Method**:

Avoid-Duplicate (see 10.1.7)
Compact-Parameter (see 10.1.9)
Object-Lock (see 10.1.38)

The **Avoid-Duplicate, Compact-Parameter** and **Object-Lock** arguments shall be applied in a manner consistent with the rules of the **Access-Method** (see Clause 11) to ensure reliable encoding. If there is an inconsistency within the application command, then it shall be considered to be in error and the encoding process shall not be implemented.

If the **Access-Method** = **Packed-Objects**, the input states are all defined in the **Packed-Objects-Constructs** argument specified in ISO/IEC 15961-1 and apply to a single Packed Object. That argument in turn has the following arguments, for which the processes are defined in Annex I:

Block-Align-Packed-Objects
Editable-Pointer-Size
ID-Type
Object-Offsets-Multiplier
Packed-Object-Directory-Type
PO-Directory-Size
PO-ID-Table
PO-Index-Length

If the **Access-Method** = **Tag-Data-Profiles**, the input states are all defined by the **Tag-Data-Profile-ID-Table** and encoded as a single Tag Data Profile using the processes defined in Annex N.

The following tag-related arguments, if declared by the Extended DSFID are used as follows to complete the encoding process:

— If the memory **Memory-Capacity** (see 9.2.10) is encoded, or the air interface has its own means of providing this information, it can be used to assess in there is sufficient memory for the command to be fully implemented.

— If one or both **Data-CRC-Indicators** (see 9.2.11) are set, then the CRC-16 shall be applied as defined in 9.2.12.

— If the **Length-Of-Encoded-Data** (see 9.2.10) is signalled to be encoded, then this shall be calculated after all encoded bytes have been determined. This field is then written, or overwritten, taking into account the availability of **DSFID-Pad-Bytes** (see 9.2.17).

If the compound argument **Multiple-Records-Constructs** is used in the command, then the **Object-Identifier** structure as define in 11.5 shall be used for identifying each data **Object** in the **add-Objects-List** argument. If this is not the case, then the command shall not be processed. In addition to writing the Multiple-Record itself, this command can require other areas of the Logical Memory to be encoded: The MR-header if the number of records is maintained and the directory if already encoded or called for by the command arguments. If any part of the memory required to implement the command is locked, then the command shall be aborted.

Encoding shall be as defined in Annex Q if the first four arcs are **1.0.15961.401**, or as defined in Annex R if the first four arcs are **1.0.15961.402** or **1.0.15961.403**. The compound argument in turn has the following arguments, for which the processes are defined in Annex Q and Annex R.

Append-To-Existing-Multiple-Record (see 10.1.5)
Application-Defined-Record-Capacity (see 10.1.6)
Identifier-Of-My-Parent (see 10.1.17)
Lock-Directory-Entry (see 10.1.24)
Lock-Record-Preamble (see 10.1.26)
Number-In-Data-Element-List (see 10.1.36)
Record-Memory-Capacity (see 10.1.46)
Record-Type-Classification (see 10.1.48)
Update-Multiple-Records-Directory (see 10.1.58)

The data shall only be encoded in memory bank 11 of the segmented memory tag.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)
Object-Identifier (see 6.3)

The **Completion-Codes** for this command will vary based on the associated compound arguments. The complete list is:

0 No-Error
8 Singulation-Id-Not-Found
9 Object-Not-Added
10 Duplicate-Object
11 Object-Added-But-Not-Locked
25 Password-Mismatch
26 AFI-Mismatch
27 DSFID-Mismatch
29 Object-Not-Editable
31 Packed-Object-ID-Table-Not-Recognised-No-Encoding
32 Tag-Data-Profile-ID-Table-Not-Recognised
33 Insufficient-Tag-Memory
36 Command-Cannot-Process-Monomorphic-UII
37 Data-CRC-Not-Applied
38 Length-Not-Encoded-In-DSFID
43 Data-Format-Not-Compatible-Multiple-Records-Header
44 Access-Method-Not-Compatible-Multiple-Records-Header
45 Sector-Identifier-Not-Compatible-Multiple-Records-Header
46 Record-Preamble-Not-Configured
47 Record-Preamble-Not-Locked
255 Execution-Error

8.2.4.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— The encoder supports all aspects of the **Access-Method**.

— The encoder supports encoding to a segmented memory tag.

— The encoder supports a process to match the **Access-Password** in the command with that encoded on the RFID tag.

— The encoder supports all the processed defined by the Extended DSFID

— The encoder supports the associated **Completion-Codes** and **Execution-Codes**.

— If any of the first four conditions are not supported an error is signalled by returning the **Execution-Code** = 4 **Command-Not-Supported**.

8.2.4.3 Guidance for appending data

The procedure described in 8.2.3.3 applies to this command.

8.2.5 Write-EPC-UII

The **Write-EPC-UII** command specifies an EPC code to be written to the UII memory of a segmented memory tag (e.g. Memory Bank 01 of an ISO/IEC 18000-6C tag).

8.2.5.1 Process requirements

To correctly implement this command in the Data Processor, the following process arguments shall be supported:

Access-Password (see 10.1.1)
EPC-Code (see 10.1.15)
Memory-Bank-Lock (see 10.1.30)
NSI-Bits (see 10.1.35)

The **Access-Password** in the command is required to match that on the RFID tag to promote writing data to the RFID tag.

This command can be used to initially write the **EPC-Code** to the RFID tag, or to overwrite the code value. If the new code is of a shorter length, the interrogator needs to ensure that bytes representing part of the older code are removed.

NOTE The interrogator shall calculate the length bits in the Protocol Control word. If the bits are provided by the application, the interrogator shall use its calculation of the length bits in preference.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
25 Password-Mismatch
33 Insufficient-Tag-Memory
255 Execution-Error

8.2.5.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— The encoder supports encoding to a segmented memory tag.

— The encoder supports a process to match the **Access-Password** in the command with that encoded on the RFID tag.

— The encoder supports the associated **Completion-Codes** and **Execution-Codes**.

8.2.6 Write-Password-Segmented-Memory-Tag

The **Write-Password-Segmented-Memory-Tag** command does not require any encoding, other than a transfer using the device interface (e.g. as in ISO/IEC 24791-5) to the interrogator. If it is incorporated in the Data Processor, then the output interface will simply carry through the command arguments.

8.2.6.1 Process requirements

The following process arguments shall be supported:

Password (see 10.1.40)
Password-Type (see 10.1.41)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
26 Password-Not-Written
255 Execution-Error

8.2.6.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— The encoder supports encoding to a segmented memory tag.

— The encoder supports a process to write the password in the application command to the appropriate memory location on the RFID tag.

— The encoder supports the associated **Completion-Codes** and **Execution-Codes**.

8.2.7 Write-Segments-6TypeD-Tag

This command shall not be used to write a **Monomorphic-UII**. If the **AFI** on the RFID tag declares that it is registered for a **Monomorphic-UII**, the appropriate error shall be returned and the encoding process aborted. The correct command to use is defined in 8.2.8.

The **Write-Segments-6TypeD-Tag** is used to write data to the UII segment, the item-related segment or both segments. The command may be implemented to write initial data to the ISO/IEC 18000-6 Type D RFID tag, or to add data to the tag.

The **Write-Segments-6TypeD-Tag** command is similar to the **Write-Objects-Segmented-Memory-Tag** command except that it is intended to write data to the ISO/IEC 18000-6 Type D tag that has its segmented memory boundaries determined dynamically by the encoder. The command is supported by the following compound arguments, for which individual arguments are listed below:

Ext-DSFID-Constructs
Item-Related-Add-Objects-List
Item-Related-DSFID-Constructs
Multiple-Records-Constructs
Packed-Object-Constructs
UII-Add-Objects-List
UII-DSFID-Constructs

8.2.7.1 Process requirements

To correctly implement this command in the Data Processor, the process arguments and compound arguments defined for the command in ISO/IEC 15961-1 shall be supported. For Multiple Records they are applied once per record, otherwise they are applied once per Logical Memory.

AFI (see 10.1.3)
DSFID-Lock (see 10.1.12)
Item-Related-DSFID (see 10.1.20)
Lock-UII-Segment-Arguments (see 10.1.27)
Memory-Segment (see 10.1.31)
Tag-Data-Profile-ID-Table (see 10.1.53)
UII-DSFID (see 10.1.56)

The ISO/IEC 18000-6 Type D does not require to implement any of the features of the Extended DSFID that have an impact on encoding processes. The tag has its own methods for declaring memory capacity amd lemgth of the encoded data. It also calculates a CRC-16 for each segment. Given these hardware features, this command defines no processes associated with the Extended DSFID. The Extended DSFID is still required to encode an **Access-Method** code value greater than 3.

The main determinant for compliant encoding is the **Access-Method**, which is incorporated as part of the **DSFID**. The **AFI** and **DSFID** are provided as arguments for use in one of the following ways:

— If data is being written to a blank segment, then these arguments are provided as part of this command to minimise communications.

— Depending on the memory segment concerned, if data is being added to the segmented memory, then the **AFI** and **DSFID** in the command should match the **AFI** and **DSFID** already encoded on the RFID tag, else there is an error and the encoding process can cease before significant amounts of data have been processed.

To correctly implement this command in the Data Processor, the following process arguments shall be supported for each **Object-Identifier** and **Object pair**, irrespective of the **Access-Method**:

Avoid-Duplicate (see 10.1.7)
Compact-Parameter (see 10.1.9)
Object-Lock (see 10.1.38)

The **Avoid-Duplicate, Compact-Parameter** and **Object-Lock** arguments shall be applied in a manner consistent with the rules of the **Access-Method** (see Clause 11) to ensure reliable encoding. If there is an inconsistency within the application command, then it shall be considered to be in error and the encoding process shall not be implemented.

If the **Access-Method** = **Packed-Objects**, the input states are all defined in the **Packed-Objects-Constructs** argument specified in ISO/IEC 15961-1 and apply to a single Packed Object. That argument in turn has the following arguments, for which the processes are defined in Annex I:

Block-Align-Packed-Objects
Editable-Pointer-Size
ID-Type
Object-Offsets-Multiplier
Packed-Object-Directory-Type
PO-Directory-Size
PO-ID-Table
PO-Index-Length

If the **Access-Method** = **Tag-Data-Profiles**, the input states are all defined by the **Tag-Data-Profile-ID-Table** and encoded as a single Tag Data Profile using the processes defined in Annex N.

If the compound argument **Multiple-Records-Constructs** is used in the command, then the **Object-Identifier** structure as define in 11.5 shall be used for identifying each data **Object** in the **add-Objects-List** argument. If this is not the case, then the command shall not be processed. In addition to writing the Multiple-Record itself, this command can require other areas of the Logical Memory to be encoded: The MR-header if the number of records is maintained and the directory if already encoded or called for by the command arguments. If any part of the memory required to implement the command is locked, then the command shall be aborted.

Encoding shall be as defined in Annex Q if the first four arcs are **1.0.15961.401**, or as defined in Annex R if the first four arcs are **1.0.15961.402** or **1.0.15961.403**. The compound argument in turn has the following arguments, for which the processes are defined in Annex Q and Annex R.

Append-To-Existing-Multiple-Record (see 10.1.5)
Application-Defined-Record-Capacity (see 10.1.6)
Identifier-Of-My-Parent (see 10.1.17)
Lock-Directory-Entry (see 10.1.24)
Lock-Record-Preamble (see 10.1.26)
Number-In-Data-Element-List (see 10.1.36)
Record-Memory-Capacity (see 10.1.46)
Record-Type-Classification (see 10.1.48)
Update-Multiple-Records-Directory (see 10.1.58)

The data shall only be encoded in the item-related segment of the tag memory.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command will vary based on the associated compound arguments. The complete list is:

0 No-Error
7 Object-Locked-Could-Not-Modify
8 Singulation-Id-Not-Found
9 Object-Not-Added
10 Duplicate-Object
11 Object-Added-But-Not-Locked
13 Object-Identifier-Not-Found
21 Object-Not-Modified
22 Object-Modified-But-Not-Locked
26 AFI-Mismatch
27 DSFID-Mismatch
29 Object-Not-Editable
31 Packed-Object-ID-Table-Not-Recognised-No-Encoding
32 Tag-Data-Profile-ID-Table-Not-Recognised
33 Insufficient-Tag-Memory
36 Command-Cannot-Process-Monomorphic-UII
37 Data-CRC-Not-Applied
38 Length-Not-Encoded-In-DSFID
43 Data-Format-Not-Compatible-Multiple-Records-Header
44 Access-Method-Not-Compatible-Multiple-Records-Header
45 Sector-Identifier-Not-Compatible-Multiple-Records-Header
46 Record-Preamble-Not-Configured
47 Record-Preamble-Not-Locked
255 Execution-Error

8.2.7.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

- The encoder supports all aspects of the **Access-Method**.
- The encoder supports encoding to an ISO/IEC 18000-6 Type D segmented memory tag.
- The encoder supports the associated **Completion-Codes** and **Execution-Codes**.
- If any of the first two conditions are not supported an error is signalled by returning the **Execution-Code** = 4 **Command-Not-Supported**.

8.2.7.3 Guidance for appending data

The procedure described in 8.2.3.3 applies to this command.

8.2.8 Write-Monomorphic-UII

The **Write-Monomorphic-UII** command instructs the Data Processor to write a **Monomorphic-UII**, either initially or to modify an existing **Monomorphic-UII**. Arguments are applied selectively depending on the type of RFID tag being addressed.

The generic encoding process calls for the Data Processor to check that the **AFI** in the command matches with an one for a **Monomorphic-UII** on the ISO/IEC 15961-2 Data Constructs register. If a match is not possible either because the **AFI** is not registered for a **Monomorphic-UII** or that no matching **AFI** can be found on the register then the encoding process is aborted.

If the **AFI** matches, the **Object-Identifier** is also checked for a match with that on the ISO/IEC 15961-2 Data Constructs register. A mismatch generates an appropriate **Completion-Code**, but this should only treated as a warning about constructing the command. The process continues because the **Object-Identifier** is not encoded. The Data Processor uses the explicitly defined compaction scheme associated with the **AFI** on the ISO/IEC 15961-2 Data Constructs register to encode the **Monomorphic-UII**.

8.2.8.1 Process requirements

To correctly implement this command in the Data Processor, the following process arguments shall be supported once per tag:

Access-Password (see 10.1.1)
AFI (see 10.1.3)
AFI-Lock (see 10.1.4)
Compact-Parameter (see 10.1.9)
Lock-UII-Segment-Arguments (see 10.1.27)
Memory-Bank-Lock (see10.1.30)
Memory-Type (see 10.1.32)
Object-Lock (see 10.1.38)

The command argument **Memory-Type** (see 10.1.32) should match the type of RFID tag being encoded. If there is a match, then one of the following processes is applied.

For an **ISO/IEC 18000-6 Type C tag:**

- If the **Access-Password** is in the command it is used to match that on the RFID tag to protect against unauthorised writing of data to the RFID tag.
- Next read the content of MB 01 to establish any existing encoding. If this is locked the process is aborted, otherwise the process continues.

— The encoding process compacts the **Object,** and if this results in an odd number of bytes appends the terminator byte 00_{16}.

— The Protocol Control word is constructed, incorporating the **AFI** and the length bits.

— If this command is used to over-write MB01 with a new **Monomorphic-UII** it is necessary to compare the length of the current and the new byte string. If the new code is of a shorter length, the interrogator needs to ensure that bytes representing part of the older code are over-written with zero bytes.

— If the command calls for the **Monomorphic-UII** to be locked, the entire MB01 is locked.

For an **ISO/IEC 18000-6 Type D** tag:

— Read the content of the UII segment to establish any existing encoding. If this is locked the process is aborted, otherwise the process continues.

— The encoding process compacts the **Object,** and if this results in an odd number of bytes appends the terminator byte 00_{16}.

— The Protocol Control word is constructed, incorporating the **AFI** and the length bits. If the tag has data encoded in the item related segment, or has a simple sensor, these characteristics need to be encoded and maintained in the Protocol control word.

— If this command is used to over-write the UII segment with a new **Monomorphic-UII** it is necessary to compare the length of the current and the new byte string. If the new code is of a different length (i.e. shorter or longer) to an existing **Monomorphic-UII**, the interrogator needs to check if an item related segment is already encoded. If the encoding can still be encoded in the UII segment without re-wrting the item-related segment, the encoded bytes are transferred to the interrogator. If the UII segment needs to be increased in size and none of the item-related bytes are locked, the entire memory can be re-written, otherwise the process has to be aborted.

NOTE The interrogator calculates a CRC-16 and places this at the end of the encoded words of the UII segment. Depending on the end position of the UII segment and the lock boundary, one or more pad words can be added by the interrogator to achieve alignment.

— If the command calls for the **Monomorphic-UII** to be locked, then the locking shall be as defined in the **Lock-UII-Segment-Arguments.**

If the RFID tag has a single memory for all data:

— The first requirement is to read the AFI on the RFID tag (which can be encoded in a separate memory area). If this matches continue otherwies report an AFI mismatch.

— Read at least 16 bytes of the content of the user memory to establish any existing encoding. If encoded bytes are found, then the process continues until a string of four zero bytes is found. If either the **AFI** or any part of the user memory is locked the process is aborted, otherwise the process continues.

— The encoding process compacts the **Object,** and adds the length of the compacted **Monomorphic-UII** as a prefix.

— If this command is used to over-write a new **Monomorphic-UII** it is necessary to compare the length of the current and the new byte string. If the new code is of a shorter length, the interrogator needs to ensure that bytes representing part of the older code are over-written with zero bytes.

— If the correct **AFI** for the **Monomorphic-UII** is not already encoded on the RFID tag, then this is encoded.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
7 Object-Locked-Could-Not-Modify
8 Singulation-Id-Not-Found
22 Object-Modified-But-Not-Locked
25 Password-Mismatch
26 AFI-Mismatch
33 Insufficient-Tag-Memory
34 AFI-Not-For-Monomorphic-UII
35 Monomorphic-UII-OID-Mismatch
255 Execution-Error

8.2.8.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— The **Monomorphic-UII** shall be compacted to the rules defined for the **AFI** on the ISO/IEC 15961-2 Data constructs register.

— The encoding on the tag shall be as defined in the processes above for the **Memory-Type** defined in the command.

8.2.9 Configure-Extended-DSFID

Whereas the **Configure-DSFID** command (see 8.2.2) provides the Data Processor with the basic encoding rules to follow when encoding data, there have been additional requirements that call for am extended set of encoding instructions. These include providing the Data Processor with information about the memory capacity and length of encoded data where this cammot be provided by a mechanism in the tag architecture or air interface protocol. Information is also provided about the requirement to encode a data-related CRC, and to declare to the decoder that this process has been applied. Other signal are also included and the feature can be expanded in the future.

The **Configure-Extended-DSFID** command is used prior to encoding the set these encoding parameters and to also encode the base **DSFID**.

8.2.9.1 Process requirements

To correctly implement this command in the Data Processor, the following process arguments shall be supported once per tag:

Battery-Assist-Indicator (see 9.2.15)
Data-CRC-Indicator (see 9.2.11)
DSFID-Lock (see 10.1.12)
DSFID-Pad-Bytes (see 9.2.17)
Full-Function-Sensor-Indicator (see 9.2.16)
Length-Of-Encoded-Data (see 9.2.10)
Memory-Capacity (see 9.2.10)
Memory-Length-Encoding (see 9.2.10)
Simple-Sensor-Indicator (see 9.2.14)

The Extended DSFID is encodedin slighly different ways depending on the tag architecture:

— For tags where the **DSFID** is encoded as the first byte of memory, the entire Extended DSFID is encoded starting at the position of the single **DSFID**.

— For tags that have a dedicated DSFID memory, the first byte(s) of the Extended DSFID are encoded in the dedicated memory location. The remaining bytes are then encoded from the first position of the data memory.

If there is an intention to encode some extended DSFID parameters initially and add to this later, then sufficient pad bytes need to be reserved for this purpose.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
4 DSFID-Not-Configured
5 DSFID-Not-Configured-Locked
6 DSFID-Configured-Lock-Failed
8 Singulation-Id-Not-Found
37 Data-CRC-Not-Applied
38 Length-Not-Encoded-In-DSFID
255 Execution-Error

8.2.9.2 Conformance requirements

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— If the RFID tag(s) supported by the encoder have the **DSFID** in a memory location requiring an explicit air interface command, then the Extended DSFID shall be split with the first byte(s) transported by the explicit air interface command, and the remaining bytes tramsferred with a single write transactiom.

— If the RFID tag(s) supported by the encoder have the **DSFID** in a memory location requiring a generic air interface write command, then the Extended DSFID shall be written with a single write transaction to a specific location starting with the DSFID position determined by the tag driver.

— If the encoder supports writing the **DSFID**, then it shall also support the transfer of any **DSFID-Lock** argument so that it can be implemented in the air interface.

— The encoder shall support the associated **Completion-Codes** and **Execution-Codes**.

8.2.10 Configure-Multiple-Records-Header

Encoding multiple records needs to be undertaken in a precise sequence of stages. The first stage is to configure the MR-header, as defined in this clause. The **Configure-Multiple-Records-Header** command is used to create all the fields on the RFID tag within the MR-header that provides information about the types of record encoded on the RFID tag, including the directory. The command is supported by the following compound arguments defined in ISO/IEC 15961-1, for which additional inherent arguments are also defined.

DSFID-Constructs
Ext-DSFID-Constructs

8.2.10.1 Process requirements

To correctly implement this command in the Data Processor, the process arguments and compound arguments defined for the command in ISO/IEC 15961-1 shall be supported. The application command has the following arguments:

Access-Password (see 10.1.1)
Directory-Length-EBV8-Indicator (see 10.1.13)
Lock-Multiple-Records-Header (see 10.1.25)
Multiple-Records-Features-Indicator (see 10.1.34)
Pointer-To-Multiple-Records-Directory (see 10.1.43)
Sector-Identifier (see10.1.49)
Singulation-Id (see 9.2.1)

For RFID tags with segmented memory, the multiple records shall only be encoded in the user memory. Specifically, for ISO/IEC 18000-6 Type C this is Memory Bank 11; and for ISO/IEC 18000-6 Type D this is the item-related segment.

The bit setting for sensors and batteries shall only be applied if the tag has no hardware means of defining that a sensor is attached. Currently, ISO/IEC 18000-6 Type C and Type D are the only RFID tags that support sensors. Both use hardware mechanisms to declare the presence of sensors and batteries.

The Extended DSFID component of the header is encoded in slightly different ways depending on the tag architecture:

— For tags where the **DSFID** is encoded as the first byte of memory, the entire Extended DSFID is encoded starting at the position of the single **DSFID**.

— For tags that have a dedicated DSFID memory, the first byte(s) of the Extended DSFID are encoded in the dedicated memory location. The remaining bytes are then encoded from the first position of the data memory.

The Data Processor shall accept the address at the start of the directory provided by the command, unless the address is non-existent in the tag memory. In this case, an error shall be reported.

Because the memory length encoding on the data CRC indicator refers to the requirement of these in the directory, no processing can take place whilst creating the header record.

The header shall be locked according to the instructions provided in the command.

The response arguments, as defined for the command response in ISO/IEC 15961-1, shall be supported by the Data Processor when responding to the application.

The **Completion-Codes** for this command will vary based on the associated compound arguments. The complete list is:

0 No-Error
4 DSFID-Not-Configured
8 Singulation-Id-Not-Found
25 Password-Mismatch
37 Data-CRC-Not-Applied
38 Length-Not-Encoded-In-DSFID
39 Multiple-Records-Header-Not-Configured
40 Multiple-Records-Header-Not-Locked
41 File-Support-Indicators-Not-Configured
42 File-Support-Indicators-Not-Locked
255 Execution-Error

8.2.10.2 Conformance requirements

8.2.10.2.1 DSFID conformance

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— If the RFID tag(s) supported by the encoder have the **DSFID** in a memory location requiring an explicit air interface command, then the Extended DSFID shall be split with the first byte(s) encoded by the explicit air interface command in the dedicated DSFID memory on the tag, and the remaining bytes encoded in the user memory.

— If the RFID tag(s) supported by the encoder have the **DSFID** in a memory location requiring a generic air interface write command, then the Extended DSFID shall be written to a specific location starting with the DSFID position determined by the tag driver.

— If the encoder supports writing the **DSFID**, then it shall also support the transfer of any **DSFID-Lock** argument so that it can be implemented in the air interface.

8.2.10.2.2 Other conformance

To conform, an encoder shall support this command and its arguments, subject to the following conditions:

— The encoder supports all aspects of encoding the header for the **Multiple-Records Access-Method**.

— The **Data-CRC-Indicator** shall be ignored for the header record itself.

— The total encoded data length (part of the Extended DSFID) refers to the size of the directory and shall be ignored for the header record itself.

— The encoder supports the associated **Completion-Codes** and **Execution-Codes**.

8.3 Application commands and responses– read

8.3.1 Read-Object-Identifiers

The **Read-Object-Identifiers** command instructs the interrogator to read all the **Object-Identifiers** from the RFID tag. This module can be used in advance of a more selective command to read a specific **Object**, or to identify duplicate **Object-Identifiers** so that a housekeeping procedure can be invoked. A valid response, if the RFID tag Logical Memory Map has no **Object-Identifiers** stored, is to return an empty **Object-Identifiers** list. Only one RFID tag shall be programmed per command to ensure that the read process is robust.

This command is not used to read **Object-Identifiers** from a Multiple Record. The functionality is incorporated in the **Read-Multiple-Records** command (see 8.3.8)

8.3.1.1 Process requirements

To correctly implement this command in the Data Processor, the following points need to be taken into account:

— An interrogator may provide all the functionality to achieve this application command. Alternatively, the interrogator may invoke a read command to capture all the encoded bytes a transfer the decode process to a higher level device.

— The different **Access-Methods** each provide a means of a faster delivery of the Object-Identifiers than reading the entire data on the tag. For example, the basic No-Directory syntax enables a decoder to "step over" the de-compaction process. Achieving this is an implementation issue for each decoder manufacturer.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)
Object-Identifier (see 6.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
255 Execution-Error

8.3.1.2 Conformance requirements

To conform, a decoder shall support this command and its arguments. In addition, it shall support the associated **Completion-Codes** and **Execution-Codes**.

8.3.2 Read-Logical-Memory-Map

The **Read-Logical-Memory-Map** command is intended to read the entire memory content of the RFID tag, and respond with this in a completely unstructured way (i.e. by returning the encoded byte values). No processing takes place through the Data Processor as part of this read command, so it is not possible to identify the encoding structure.

8.3.2.1 Process requirements

The command applies equally to all **Access-Methods**, but if a **Directory** structure has been defined by the **Access-Method,** this shall be included in the response, but shall not be distinguished from other bytes in the Logical Memory Map.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)
Logical-Memory-Map (see list in Clause 10)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
19 Read-Incomplete
255 Execution-Error

8.3.2.2 Conformance requirements

To conform, a decoder shall support this command. In addition, it shall support the associated **Completion-Codes** and **Execution-Codes**.

8.3.3 Read-Objects

The **Read-Objects** command is intended to read one or more data objects from the RFID tag, as defined by their **Object-Identifiers**. The argument **Read-Type** determines whether the first, selected, or all **Object-Identifier(s)** are read from the RFID tag. The command also supports an argument, **Max-App-Length**, which enables the application to prescribe an upper address point on the RFID tag beyond which reading is discontinued.

This command is not used to read **Object-Identifiers** from a Multiple Record. The functionality is incorporated in the **Read-Multiple-Records** command (see 8.3.8)

8.3.3.1 Process requirements

To correctly implement this command in the Data Processor, the following process arguments shall be supported:

Check-Duplicate (see 10.1.8)
Max-App-Length (see 10.1.28)
Read-Type (see 10.1.45)

The following arguments shall be supported for the response:

Compact-Parameter (see 10.1.9)
Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)
Lock-Status (see list in Clause 10)
Object (see 6.4)
Object-Identifier (see 6.3)

Except for **Execution-Code**, the response arguments are associated with each data element.

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
10 Duplicate-Object
13 Object-Identifier-Not-Found
15 Object-Not-Read
255 Execution-Error

8.3.3.2 Conformance requirements

To conform, a decoder shall support this command and its arguments, subject to the following:

— If the **Check-Duplicate** argument is set to TRUE, then the entire encoded data shall be read to establish that the subject **Object-Identifier** is not duplicated. If the **Access-Method** is **Multiple-Records**, then the **Check-Duplicate** argument is only applied to the specific record being accessed. As there is no requirement to decode the data **Object** for **Object-Identifiers** not called out in the application command, a process similar to the **Read-Object-Identifiers** command could be used.

— If the **Max-App-Length** argument is invoked, the interrogator and decoder only need to process the bytes up to the value for that argument.

— The decoder shall support the associated **Completion-Codes** and **Execution-Codes**.

8.3.4 Inventory-ISO-UIImemory

The **Inventory-ISO-UIImemory** command is intended to return the contents of the UII memory from a number of segmented memory tags, given the expectation that an **Object-Identifier** for a non-**EPC-Code** is encoded. The response returns the content of the UII memory for all tags whose encoded bit string matches the arguments of the command.

8.3.4.1 Process requirements

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Additional-App-Bits (see 10.1.2)
AFI (see 10.1.3)
DSFID (see 10.1.11)

The arguments provided in the command enable a bit mask to be incorporated into appropriate air interface protocol commands to select only tags that match the bit mask. For tags with encoding based on **Object-Identifier**, the **AFI** is included as a minimum component of the mask. The **DSFID** may be added, particularly where its value is constant for an application. Additional bits may be added that, for example, identify the Precursor of the first (sometimes only) encoded **Data-Set** and any additional bits that a certain to create a tag selection criterion.

The Data Processor concatenates these values into a contiguous bit stream, and prepends this with a bit = '1' and identifies this bit stream as beginning at Bit 17h of Memory Bank 01. An error occurs if Bit 17h of Memory Bank 01 = '0'.

The following arguments shall be supported for the response:

Compact-Parameter (see 10.1.9)
Completion-Code (see 10.2 and below)
DSFID (see 10.1.11)
Execution-Code (see 10.3)
Lock-Status (see list in Clause 10)
Object (see 6.4)
Object-Identifier (see 6.3)
Protocol-Control-Word (see list in Clause 10)

If the UII memory encodes more than one **Object-Identifier**, then these arguments shall be returned for each data element: **Compact-Parameter, Completion-Code** (some values), **Lock-Status, Object**, and **Object-Identifier**.

The **Completion-Codes** for this command are:

0 No-Error
10 Duplicate-Object
13 Object-Identifier-Not-Found
15 Object-Not-Read
255 Execution-Error

8.3.4.2 Conformance requirements

To conform, a decoder shall support this command and its arguments, subject to the following conditions:

— The decoder supports decoding from a segmented memory tag.

— The decoder supports the associated **Completion-Codes** and **Execution-Codes**.

8.3.5 Inventory-EPC-UIImemory

The **Inventory-EPC-UIImemory** command is intended to return the contents of the UII memory from a number of segmented memory tags, given the expectation that an **EPC-Code** is encoded. The response returns the content of the UII memory for all tags whose encoded bit string matches the arguments of the command.

8.3.5.1 Process requirements

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Length-of-Mask (see 10.1.23)
Pointer (see 10.1.42)
Tag-Mask (see 10.1.54)

If the value of the pointer is less than 18h, an error occurs if bit 17h of Memory Bank 01 = '1'.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
EPC-Code (see 10.1.15)
Execution-Code (see 10.3)
Protocol-Control-Word (see list in Clause 10)

The **Completion-Codes** for this command are:

0 No-Error
255 Execution-Error

8.3.5.2 Conformance requirements

To conform, a decoder shall support this command and its arguments, subject to the following conditions:

— The decoder supports decoding from a segmented memory tag.

— The decoder supports the associated **Completion-Codes** and **Execution-Codes**.

8.3.6 Read-Words-Segmented-Memory-Tag

The **Read-Words-Segmented-Memory-Tag** command instructs the interrogator to read a contiguous sequence of words from one of the memory banks of a segmented memory RFID tag. This command can be used to extract encoded bytes, which might not be object-based such as the unique **Singulation-Id** or a password. It can also be useful for diagnostic purposes.

8.3.6.1 Process requirements

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Access-Password (see 10.1.1)
Memory-Bank (see 10.1.29)
Word-Count (see 10.1.59)
Word-Pointer (see 10.1.60)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)
Read-Data (see list in Clause 10)

The **Completion-Codes** for this command are:

0 No-Error
25 Password-Mismatch
255 Execution-Error

8.3.6.2 Conformance requirements

To conform, a decoder shall support this command and its arguments, subject to the following conditions:

— The decoder supports decoding from a segmented memory tag.

— The decoder supports a process to match the **Access-Password** in the command with that encoded on the RFID tag.

— The decoder supports the associated **Completion-Codes** and **Execution-Codes**.

8.3.7 Read-Segments-6TypeD-Tag

The **Read-Segments-6TypeD-Tag** command instructs the interrogator to read all the data from an ISO/IEC 18000-6 Type D tag and instructs the Data Processor to provide the decoded data according to the arguments in the command. This command can be used to show the content of each segment as a raw byte string, or to provide more detailed analysis of the content of one or more segments.

This command is not used to read **Objects** or **Object-Identifiers** from a Multiple Record encoded on an RFID tag compliant with ISO/IEC 18000-6 Type D. The functionality is incorporated in the **Read-Multiple-Records** command (see 8.3.8)

8.3.7.1 Process requirements

To correctly implement this command in the Data Processor, the following argument shall be supported:

Segment-Read-Type (see 10.1.50)

The ISO/IEC 18000-6 type D air protocol delivers the entire data payload of the RFID. Therefore, the command argument codes specify the processes that the Data Processor undertakes, as defined in 10.1.50.

The following arguments shall be supported for the response, and be included depending on the command arguments:

Compact-Parameter (see 10.1.9)
Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)
Item-Related-DSFID (see 10.1.20)
Item-Related-Segment-Map (see 10.1.21)
Lock-Status (see list in Clause 10)
Object (see 6.4)
Object-Identifier (see 6.3)
Simple-Sensor-Data-Block (see 10.1.51)
TID-Segment-Map (see 10.1.55)
UII-Segment-Map (see 10.1.57)

The **Completion-Codes** for this command are:

0 No-Error
19 Read-Incomplete
255 Execution-Error

8.3.7.2 Conformance requirements

To conform, a decoder shall support this command and its arguments, subject to the following conditions:

— The decoder supports decoding from to a segmented memory tag compliant with ISO/IEC 18000-6 Type D.

— The decoder supports the processing into segments and further into the constituent parts of the UII segment and the Item-related segment

— The decoder supports the associated **Completion-Codes** and **Execution-Codes**.

8.3.8 Read-Multiple-Records

The **Read-Multiple-Records** command instructs the interrogator to read various logically structured components from an RFID tag configured to encode multiple records. This includes performing relevant functions defined in 8.3.1 to read a set of one or more **Object-Identifiers**; and performing the relevant functions defined in 8.3.3 to read data **Objects** from an individual record.

The command is applied to memory bank 11 of the ISO/IEC 18000-6 Type C tag, and to the Item-related data segment of the ISO/IEC 18000-6 Type D tag. Only one RFID tag shall be programmed per command to ensure that the reading process is robust.

The command is supported by the following compound argument defined in ISO/IEC 15961-1, for which the additional inherent **Read-Objects** argument is also defined.

8.3.8.1 Process requirements

To correctly implement this command in the Data Processor, the process arguments and compound arguments defined for the command in ISO/IEC 15961-1 shall be supported. The application command has the following arguments:

Check-Duplicate (see 10.1.8)
Max-App-Length (see 10.1.28)
Object-Identifier (see 6.3)
Read-Record-Type (see 10.1.44)
Singulation-Id (see 9.2.1)

The **Read-Record-Type** codes 3 to 9 (as defined in 10.1.44) require the use of one or more **Object-Identifiers** in the command for the Data Processor to invoke the relevant processes. Three very specific formats of **Object-Identifier** are applied to multiple records:

— For a multiple record that is not part of a hierarchy, the structure is: 1.0.15961.401.{Data-Format}.{sector identifier}.{record type}.{instance-of}.{Relative-OID of data element}

— For a multiple record that is part of a hierarchy, but not a data element list, the structure is: **1.0.15961.402.{Data-Format}.{sector identifier}.{record type}.{hierarchical id}.{Relative-OID of data element}**

— For a multiple record that is a data element list, the structure is: 1.0.15961.403.{Data-Format}.{sector identifier}.{record type}.{hierarchical id}.{Relative-OID of data element}

NOTE This object identifier calls for response to include all the list element number.

The first two **Object-Identifier** structures apply to **Read-Record-Type** codes 3 to 8. The third listed **Object-Identifier** structure only applies to **Read-Record-Type** codes 3, 4 and 9.

If **Read-Multiple-Records-Header** (**Read-Record-Type** code 0) is selected, the Data Processor returns the interpretation of the MR-header in the **Multiple-Records-Header-Structure** argument. The response contains the following compound arguments, for which individual arguments are listed below:

— **DSFID-Constructs** with the following inherent arguments:

Access-Method (see Clause 11)

Data-Format (see 9.2.5)

— **Ext-DSFID-Constructs** with the following inherent arguments:

Battery-Assist-Indicator (see 9.2.15)
Data-CRC-Indicator (see 9.2.11)
DSFID-Pad-Bytes (see 9.2.17)
Full-Function-Sensor-Indicator (see 9.2.16)

Length-Of-Encoded-Data (see 9.2.10)
Memory-Capacity (see 9.2.10)
Memory-Length-Encoding (see 9.2.10)
Simple-Sensor-Indicator (see 9.2.14)

The following arguments are also in the **Multiple-Records-Header-Structure** argument:

Multiple-Records-Directory-Length (see 10.1.33)
Multiple-Records-Features-Indicator (see 10.1.34)
Number-Of-Records (see 10.1.37)
Pointer-To-Multiple-Records-Directory (see 10.1.43)
Sector-Identifier (see 10.1.49)

If **Read-Multiple-Records-Header-Plus-1st-Preamble** (**Read-Record-Type** code 1) is selected, the Data Processor returns the interpretation of the MR-header in the **Multiple-Records-Header-Structure** argument and the interpretation of the first record's preamble in the **Multiple-Records-Preamble-Structure** argument.

The response is a combination of the responses for **Read-Record-Type** code 0 and 3). It contains the specific responses for the compound arguments **DSFID-Constructs**, **Ext-DSFID-Constructs** and **Multiple-Records-Header-Structure** that are all listed above, and the specific responses for the compound argument **Multiple-Records-Header-Structure** that are listed for the **Read-Preamble-Specific-Multiple-Record** (below).

If **Read-Multiple-Records-Directory** (**Read-Record-Type** code 2) is selected, the Data Processor returns the interpretation of the multiple records directory in the **Multiple-Records-Directory-Structure** argument. The response contains the following compound arguments, for which individual arguments are listed below:

— **DSFID-Constructs** with the inherent arguments as defined for the **Read-Multiple-Records-Header**

— **Ext-DSFID-Constructs** with the inherent arguments as defined for the **Read-Multiple-Records-Header**

The following arguments are also in the **Multiple-Records-Directory-Structure** argument:

Hierarchical-Identifier-Arc (see 10.1.16)
Identifier-Of-My-Parent (see 10.1.17)
Instance-Of-Arc (see 10.1.19)
Record-Type-Arc (see 10.1.47)
Record-Type-Classification (see 10.1.48)
Sector-Identifier (see 10.1.49)
Start-Address-Of-Record (see 10.1.52)

If **Read-Preamble-Specific-Multiple-Record** (**Read-Record-Type** code 3) is selected, the command shall include a single **Object-Identifier** in the **Read-Objects List** that is certainly defined down to the record type arc, and the instance of arc (if applicable) or the hierarchical id arc (if applicable). he Data Processor returns the interpretation of the record's preamble in the **Multiple-Records-Preamble-Structure** argument. The response contains the following compound arguments, for which individual arguments are listed below:

— **DSFID-Constructs** with the inherent arguments as defined for the **Read-Multiple-Records-Header**

— **Ext-DSFID-Constructs** with the inherent arguments as defined for the **Read-Multiple-Records-Header**

The following arguments are also in the **Multiple-Records-Preamble-Structure** argument:

Data-Length-Of-Record (see 10.1.10)
Encoded-Memory-Capacity (see 10.1.13)
Hierarchical-Identifier-Arc (see 10.1.16)
Identifier-Of-My-Parent (see 10.1.17)
Instance-Of-Arc (see 10.1.19)
Record-Type-Arc (see 10.1.47)

Record-Type-Classification (see 10.1.48)
Sector-Identifier (see 10.1.49)
Start-Address-Of-Record (see 10.1.52)

The **Read-All-Record-OIDs-Specific-Record-Type** (**Read-Record-Type** code 4) is used for identifying a series of history records of the same type or a set of records in a hierarchy of the same type. The command shall include a single **Object-Identifier** in the **Read-Objects List** that is only defined down to the record type arc. The Data Processor returns the list of **Object-Identifiers** one layer lower in the **Read-OIDs-Response-List**, i.e. either with the set of instance-of arcs or with the set of hierarchical id arcs. To achieve this, the Data Processor needs to parse the entire Multiple Records directory; or if this is not encoded, the Data Processor read and parse the preamble of each Multiple Record. During this procedure, the Data Processor shall only retain and report on records whose record type arc matches that in the **Object-Identifier** in the command.

If **Read-OIDs-Specific-Multiple-Record** (**Read-Record-Type** code 5) is selected, the command shall include a single **Object-Identifier** in the **Read-Objects List** that is certainly defined down to the record type arc, and the instance-of arc (if applicable) or the hierarchical id arc (if applicable). The Data Processor returns the list of **Object-Identifiers** encoded within the record in the **Read-OIDs-Response-List**. This does not apply to Data Element lists.

If **Read-All-Objects-Specific-Multiple-Record** (**Read-Record-Type** code 6) is selected, the command shall include a single **Object-Identifier** in the **Read-Objects List** that is certainly defined down to the record type arc, and the instance-of arc (if applicable) or the hierarchical id arc (if applicable). The Data Processor returns the list of **Object-Identifiers** and **Objects** encoded within the record in the **Read-Objects-Response-List**. This does not apply to Data Element lists.

If **Read-Multiple-Objects-Specific-Multiple-Record** (**Read-Record-Type** code 7) is selected, the command shall include the nominated **Object-Identifiers** that are defined down to the specific data element in the **Read-Objects List**. The Data Processor returns the list of nominated **Object-Identifiers** and **Objects** encoded within the record in the **Read-Objects-Response-List**. This does not apply to Data Element lists.

If **Read-1st-Objects-Specific-Multiple-Record** (**Read-Record-Type** code 8) is selected, the command shall include the nominated **Object-Identifier(s)** that are defined down to the specific data element(s) in the **Read-Objects List**. The Data Processor returns the list of nominated **Object-Identifiers** and **Objects** encoded up to the **Max-App-Length** within the record in the **Read-Objects-Response-List**. This does not apply to Data Element lists.

If **Read-Data-Element-List-Specific-Multiple-Record** (**Read-Record-Type** code 9) is selected, the command shall include a single **Object-Identifier** in the **Read-Objects List** that is defined down to the data element. The Data Processor first checks that the record is a Data Element list. If so, it uses the rules of the **Access-Method** to re-construct the **Object-Identifier** down to the list element number as encoded in the data element list and returns this and associated **Objects** encoded within the record in the **Read-Objects-Response-List**. The first four arcs of the **root-OID** for the application command and response is **1.0.15961.403**. If this is not in the command, than the Data Processor shall not implement the command.

The **Completion-Codes** for this command will vary based on the **Read-Record-Type** code value set in the command. The complete list is:

0 No-Error
8 Singulation-Id-Not-Found
10 Duplicate-Object
13 Object-Identifier-Not-found
15 Object-Not-Read
35 Monomorphic-UII-OID-Mismatch
48 Multiple-Records-Directory-Not-Present
255 Execution-Error

8.3.8.2 Conformance requirements

To conform, a decoder shall support this command and its arguments. In addition, it shall support the associated **Completion-Codes** and **Execution-Codes**.

To conform, a decoder shall support this command and its arguments, subject to the following conditions:

— All the **Read-Record-Types** shall be supported.

— For **Read-Record-Type** codes 5, 6, 7,and 8, if the **Check-Duplicate** argument is set to TRUE, then the entire record shall be read to establish that the subject **Object-Identifier** is not duplicated.

— For **Read-Record-Type** codes 4 and 9, if the **Check-Duplicate** argument is set to TRUE, then the argument shall be ignored, because the intention is to identify multiple occurrences of the **Object-Identifier** in the command and return any additional arc.

— If the **Max-App-Length** argument is invoked, the interrogator and decoder only need to process the bytes up to the value for that argument.

— The decoder shall support the associated **Completion-Codes** and **Execution-Codes.**

The decoder needs to support the following, depending on the requirements of the application:

— At least one of the **Access-Methods** defined in this International Standard

— The decoding from a segmented memory tag. compliant with ISO/IEC 18000-6 Type C.

— The decoder supports decoding from to a segmented memory tag compliant with ISO/IEC 18000-6 Type D.

8.4 Application commands and responses– other

8.4.1 Inventory-Tags

The **Inventory-Tags** command is intended to read a set of **Singulation-Ids** from RFID tags that have a particular **AFI**. It is only applicable where an air interface command supports an inventory process using the **AFI** as a named argument and, generally, where a unique chip identifier is used in the arbitration process.

8.4.1.1 Process requirements

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Identify-Method (see 10.1.18)
Number-Of-Tags (see 10.1.18)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)
Identities (see list in Clause 10)
Number-of-Tags-Found (see list in Clause 10)

Typically, the response should include the argument Identities, which is a list of **Singulation-Ids** each of which has met the selection criteria.

The **Completion-Codes** for this command are:

0 No-Error
23 Failed-To-Read-Minimum-Number-Of-Tags
24 Failed-To-Read-Exact-Number-Of-Tags
255 Execution-Error

8.4.1.2 Conformance requirements

To conform, a decoder shall support this command and its arguments, subject to the following conditions:

— That the RFID tag(s) supported by the decoder are capable of being selected using an explicitly declared AFI and that the response **Singulation-Id** is used in the arbitration process.

— The decoder shall support the associated **Completion-Codes** and **Execution-Codes**.

8.4.2 Delete-Object

The **Delete-Object** command is intended to remove an **Object-Identifier** and its associated data **Object** from the memory of the RFID tag. Invoking this command depends on the **Access-Method** declared for the RFID tag. Each of the cases is discussed under the appropriate **Access-Method** (see Clause 11).

8.4.2.1 Process requirements

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Check-Duplicate (see 10.1.8)

The manner in which the **Delete-Object** command is implemented in the Logical Memory is dependent on the **Access-Method**:

— For the **No-Directory** and **Directory Access-Methods** if other encoding follows the deleted **Object-Identifier**, the Data Processor may replace the deleted bytes with a null **Data-Set** (see Annex D.6.4). This procedure is invoked automatically by the Data Processor.

— For **Packed-Objects** the **Delete-Object** command is supported in one of two ways:

  i) If the Packed Object contains an Addendum section then the rules defined in Annex I.5.6 are applied.

  ii) Else the entire Packed Object needs to be re-written.

— For **Tag-Data-Profiles** the Data Processor shall replace the deleted bytes with a null **Data-Set** (see Annex D.6.4).

— For the **Records-Directory Access-Method**, the target **Object-Identifier** is identified according to the rules defined in 11.5.2. Within the record, the rules of the particular record-specific **Access-Method** shall be followed.

If any of the bytes associated with the **Object-Identifier** is locked, then the Data Processor shall return an error.

If the **Check-Duplicate** flag is set to TRUE, the interrogator shall verify, before deleting the requested **Object**, that there is only a single instance of the requested **Object-Identifier**. If the interrogator detects that the RFID tag is encoding more than one instance of the referenced **Object-Identifier**, it shall not perform the **Delete-Object** function and shall return the appropriate **Completion-Code**.

If the **Check-Duplicate** flag is set to FALSE, the interrogator shall delete the first occurrence of the data set specified by its **Object-Identifier**.

NOTE This is an argument that effectively provides no protection against duplicate **Object-Identifiers**. It should only be used when there is a high expectation of no duplicates.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
10 Duplicate-Object
12 Object-Not-Deleted
13 Object-Identifier-Not-Found
14 Object-Locked-Could-Not-Delete
255 Execution-Error

8.4.2.2 Conformance requirements

To conform, a device shall support encoding and decoding to support this command and its arguments, subject to the following conditions:

— The device supports a delete function as defined for the **Access-Methods** that it supports

— If the **Check-Duplicate** argument is set to TRUE, then the entire encoded data shall be read to establish that the subject **Object-Identifier** is not duplicated prior to invoking the delete process.

— The device supports the associated **Completion-Codes** and **Execution-Codes**.

If the **Access-Method** is **Multiple-Records**, then the **Check-Duplicate** argument is only applied to the specific record being accessed.

8.4.3 Modify-Object

The **Modify-Object** command is intended to change the value of a data **Object** associated with an **Object-Identifier** already encoded on the memory of the RFID tag. The complete memory needs to be read to ensure that the **Object-Identifier** is not duplicated. If so the command is aborted. Invoking this command depends on the **Access-Method** declared for the RFID tag. In addition, the procedure is different if the resultant encoding length of the modified object is different from the original length. Each of the cases is discussed under the appropriate **Access-Method** (see Clause 11).

If the data object is part of a multiple record (identified with the basic root-OID of **1.0.15961.401**) or a hierarchical multiple record (identified with the basic root-OID of **1.0.15961.402), then the process to** modify a data object shall be that of the declared **Access-Method**. A data object cannot be modified on a data element list (identified with the basic root-OID of **1.0.15961.403**). If a command carries the instructions to modify a data element list, the Data Processor shall not implement the command and report an error.

8.4.3.1 Process requirements

The **Modify-Object** command instructs the Data Processor to carry out three related processes:

1) Read the complete Logical Memory Map from the RFID tag. If the command is being applied to a Multiple Record, then only that record needs to be read.

2) Identify the encoded packet specified by the **Object-Identifier**. If duplicated instances are found, the process is aborted.

3) Over-write with the modified encoded packet, or, depending on the Access-Method,

— invoking a combination of a writing a null **Data-Set** where the original bytes were encoded and writing a new **Data-Set**; or

— invoking a combination of a deleting and writing a new encoded packet.

If any of the bytes associated with the **Object-Identifier** is locked, then the Data Processor shall abort the process and return the appropriate completion code.

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Compact-Parameter (see 10.1.9)
Object-Lock (see 10.1.38)

The manner in which the **Modify-Object** command is implemented in the Logical Memory is dependent on the **Access-Method**:

For the **Multiple-Records Access-Method** the **Modify-Object** command is supported as follows:

i) The target Object-Identifier is identified according to the rules defined in 11.5.2.

ii) Within the record, the rules of the particular record-specific Access-Method shall be followed (as defined below).

For the **No-Directory** and **Directory Access-Methods**:

— If the byte string, that represents the modified data when prepared for encoding in the Logical Memory Map, is the same length as its previous encoded byte string, the modified value is generally written to the same positions.

— If the byte string is shorter than the previous encoded byte string, then the Data Processor automatically encodes an offset to re-align the modified encoding with the original.

— If the byte string is longer than the previous encoded byte string, then it needs to be located in a different area of the Logical Memory Map with this process controlled by the Data Processor. A similar situation might arise if the **Object-Lock** argument is set to TRUE in the command. Alternatively, the Data Processor may apply a procedure that automatically replaces the modified bytes with one or more **Null-Bytes** (see Annex D.6.4) and writes the longer **Data-Set** ant the end of encoded memory.

For the **Directory Access-Method**, in addition to modifying the **Data-Set** itself (as defined above), additional processing might be necessary to update the directory.

For **Packed-Objects** the **Modify-Object** command is supported in one of two ways:

i) If the Packed Object contains an Addendum section then the rules defined in Annex I.5.6 are applied.

ii) Else the entire Packed Object needs to be re-written.

For **Tag-Data-Profiles** the following rules apply:

— If the byte string, that represents the modified data when prepared for encoding in the Logical Memory Map, is the same length as its previous encoded byte string, the modified value is written to the same positions.

— If the byte string is shorter than the previous encoded byte string, then the Data Processor automatically encodes an offset to re-align the modified encoding with the original.

— If the byte string is longer than the previous encoded byte string, then there is an error and the process shall cease.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
7 Object-Locked-Could-Not-Modify
8 Singulation-Id-Not-Found
10 Duplicate-Object
13 Object-Identifier-Not-Found
21 Object-Not-Modified
22 Object-Modified-But-Not-Locked
255 Execution-Error

8.4.3.2 Conformance requirements

To conform, a device shall support encoding and decoding to support this command and its arguments, subject to the following conditions:

— The device supports a modify function as defined for the **Access-Methods** that it supports.

— The device supports the associated **Completion-Codes** and **Execution-Codes**.

8.4.3.3 Guidance for appending modified data

If the modified data is longer that the original data and requires to be appended, the the procedure described in 8.2.3.3 applies to this command.

8.4.4 Erase-Memory

The **Erase-Memory** command is intended to reset to zero the entire **Logical-Memory-Map** of a particular RFID tag. This includes the directory if this is defined by the **Access-Method**.

8.4.4.1 Process requirements

To correctly implement this command in the Data Processor, all bytes shall be reset, except those that are locked. In this case, the **Completion-Code Blocks-Locked (17)** shall be returned.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
17 Blocks-Locked
18 Erase-Incomplete
255 Execution-Error

8.4.4.2 Conformance requirements

To conform, a device shall support encoding and decoding to support this command and its arguments.

8.4.5 Get-App-Based-System-Info

The **Get-App-Based-System-Info** command is used to return the **AFI** and **DSFID** from the RFID tag. In some air interface protocols, the equivalent air interface command is the only way in which this information can be retrieved.

8.4.5.1 Process requirements

To correctly implement this command in the Data Processor shall call up the equivalent air interface command, and transfer the **Singulation-Id** to that command.

The following arguments shall be supported for the response:

AFI (see 10.1.3)
Completion-Code (see 10.2 and below)
DSFID (see 10.1.11)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
20 System-Info-Not-Read
255 Execution-Error

8.4.5.2 Conformance requirements

To conform, a decoder shall support this command and its arguments, subject to the following conditions:

— That the air interface supported by the decoder has the equivalent command (e.g. ISO/IEC 18000-3 Mode 1).

— The decoder shall support the associated **Completion-Codes** and **Execution-Codes**.

8.4.6 Kill-Segmented-Memory-Tag

The **Kill-Segmented-Memory-Tag** command instructs the interrogator to apply appropriate air interface protocols to render the RFID tag unreadable in future. The **Kill-Password** in the command must match the Password encoded on the RFID tag.

8.4.6.1 Process requirements

The **Kill-Segmented-Memory-Tag** command does not require any processing through the Data Processor, other than a transfer using the device interface (e.g. as in ISO/IEC 24791-5) to the interrogator. If it is incorporated in the Data Processor, then the output interface will simply carry through the command arguments.

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
27 Zero-Kill-Password-Error
28 Kill-Failed
255 Execution-Error

8.4.6.2 Conformance requirements

To conform, a device shall support this command and its arguments, subject to the following conditions:

— The device supports processing with a segmented memory tag.

— The device shall support the associated **Completion-Codes** and **Execution-Codes**.

8.4.7 Delete-Packed-Object

The purpose of the **Delete-Packed-Object** command is to delete a complete Packed Object, by using an encoded **Object-Identifier** as an alias for the particular Packed Object.

8.4.7.1 Process requirements

The **Delete-Packed-Object** command instructs the Data Processor to:

1) Identify the Packed Object using the Object-Identifier.

2) Establish the memory location on the entire Packed Object.

3) Erase the complete memory typically by over-writing with the default byte value of the RFID tag.

If any part of the Packed Object is locked, then the command cannot be invoked, and the process shall be aborted and the appropriate completion code returned.

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Check-Duplicate (see 10.1.8)
Object-Identifier (see 6.3)
Singulation-Id (see 9.2.1)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
10 Duplicate-Object
12 Object-Not-Deleted
13 Object-Identifier-Not-Found
14 Object-Locked-Could-Not-Delete
255 Execution-Error

8.4.7.2 Conformance requirements

To conform, a device shall support this command and its arguments, subject to the following conditions:

— The device supports processing of **Access-Method Packed-Objects**.

— The device shall support the associated **Completion-Codes** and **Execution-Codes**.

8.4.8 Modify-Packed-Object-Structure

The **Modify-Packed-Object-Structure** command is used to change the structure of a Packed Object, for example to introduce a directory structure. An encoded **Object-Identifier** is used as an alias for the particular Packed Object.

8.4.8.1 Process requirements

The **Modify-Packed-Object-Structure** command instructs the Data Processor to:

1) Identify the Packed Object using the Object-Identifier.

2) Check the status of any directory or capability to add a directory.

3) Modify the particular Packed Object as necessary

4) If necessary based on the command arguments, write an additional Packed Object directory.

If it is not possible to add the directory structure, or one is already in place, then the process shall be aborted and the appropriate completion code returned.

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Check-Duplicate (see 10.1.8)
Object-Identifier (see 6.3)
Packed-Object-Directory-Type (see 10.1.39)
Singulation-Id (see 9.2.1)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0	No-Error
7	Object-Locked-Could-Not-Modify
8	Singulation-Id-Not-Found
10	Duplicate-Object
13	Object-Identifier-Not-Found
30	Directory-Already-Defined
255	Execution-Error

8.4.8.2 Conformance requirements

To conform, a device shall support this command and its arguments, subject to the following conditions:

— The device supports processing of **Access-Method Packed-Objects**.

— The device shall support the associated **Completion-Codes** and **Execution-Codes**.

8.4.9 Delete-Multiple-Record

The **Delete-Multiple-Record** command instructs the interrogator to mark a Multiple Record as deleted. The bytes that make up the record are not actually deleted, but the subject record and its entry in the directory (if present) have code values set to indicate that the record is no longer to be treated as valid.

8.4.9.1 Process requirements

The **Object-Identifier** is to the level where the final arc is that of the instance-of or hierarchical identifier. Multiple-Records are identified with one of the following basic **root-OIDs**:

1.0.15961.401

1.0.15961.402

1.0.15961.403

If the **Object-Identifier** has a different root, the the command shall not be processed.

If either the record preamble or the directory is locked, then the command cannot be invoked. If the record is part of a hierarchical structure with the **root-OID 1.0.15961.402**, then all the child(ren) records and in turn their children need to be deleted first. If any of these record preambles, or the associated directory entry, is locked then none of the records can be deleted. This requires the hierarchy to be established and the lock status of each record in that hierarchy to have an unlocked status in the preambles and the directory (if present).

To correctly implement this command in the Data Processor, the following arguments shall be supported:

Access-Password (see 10.1.1)
Object-Identifier (see 6.3)
Singulation-Id (see 9.2.1)

The following arguments shall be supported for the response:

Completion-Code (see 10.2 and below)
Execution-Code (see 10.3)

The **Completion-Codes** for this command are:

0 No-Error
8 Singulation-Id-Not-Found
13 Object-Identifier-Not-Found
25 Password-Mismatch
49 Record-Not-Deleted-Preamble-Locked
50 Record-Not-Deleted-Directory-Locked
51 Record-Not-Deleted-Lower-Level-Preamble-Locked
255 Execution-Error

8.4.9.2 Conformance requirements

To conform, a device shall support this command and its arguments, subject to the following conditions:

— The device supports processing of **Access-Method Multiple-Records**.

— The device shall support the associated **Completion-Codes** and **Execution-Codes**.

8.5 Air interface support for application commands

Different air interface protocols are able to support some, but not all, of the application commands. For a specific air interface protocol, an application command:

— Can be supported as included in this International Standard, whether this is because there is with a directly equivalent air interface command, or by invoking a series of air interface commands.

— Is unable to be supported, because particular features are not supported on the air interface protocol, even in an optional manner.

— Can be supported, but another application command achieves the same results probably in a better way.

Annex A provides this information in more detail for specific air interface protocols.

9 Data Processor and the air interface

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

9.1 Air interface services

This International Standard is open-ended with respect to the fact that new types of RFID tag may be added to the ISO/IEC 18000 multi-part standard that require some interface with the Data Processor. To achieve this, some basic presumptions are made about the types of RFID 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.

NOTE The term 'block' is used in this International Standard in a manner consistent with its definition in Clause 3, because this aligns with the most common usage of the term in air interface protocols. In ISO/IEC 18000-6C the term 'word' has a similar function.

— 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 RFID tag in ISO/IEC 18000 shall provide the following air interface services of the physical characteristics and data capabilities of the RFID 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 RFID 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 and/or addressing of groups of RFID tags using to the AFI as part of the selection mechanism.

6) Provide a mechanism to write and read the DSFID.

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

8) Optionally provide the writing capabilities supported by the RFID tag.

9) Optionally provide the locking capabilities supported by the RFID 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 B. Details of specific Tag Drivers are provided in Annex C.

9.2 Defining the system information

The system information is a set of elements that is encoded, or provided by other means, from the RF tag to the Data Processor when communications are established. The systems information shall consist of the elements listed in Table 4 — System information elements with the length of each element, and shall be as defined in the following subclasses.

Table 4 — System information elements

Element	Length
Singulation-Id	up to 255 bytes
physical block size (in bytes)	1 byte (hexadecimal value of block size)
number of blocks	1 or more bytes (hexadecimal value of number of blocks)
AFI	1 byte
DSFID	1 byte

9.2.1 Singulation-Id

The **Singulation-Id** is the means of ensuring reliable communication between the application and the RFID tag throughout the transaction process via the Data Processor. It is communicated across the air interface and the application interface. The **Singulation-Id** may be up to 256 bytes long, depending on the rules of the air interface protocol or the way that the interrogator maintains a register of RFID tags with which it communicates.

NOTE In the 2004 edition of this International standard the Singulation-Id was called the 'Tag-Id'. Apart from the change of name all functions remain the same.

The format of the **Singulation-Id** shall be as defined for each Tag Driver (see Annex C) and shall be based on one of the following:

a) A completely unique identifier programmed in the RFID 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/IEC 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 Singulation-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 identifier across the air interface, but requiring the data related identifier to be returned as a response.

9.2.2 Physical block size

The physical block size is defined as the minimum number of 8-bit bytes that are capable of being written to, read from, on the particular RFID tag. If the minimum number of bytes differs for read and write operations, for the purposes of this International Standard the block size shall be the smaller value. The value is constrained to between 1 and 256 bytes.

NOTE The block size relates to memory units on the RFID 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 Processor. The physical block size is not communicated across the application interface.

9.2.3 Number of blocks

The number of blocks is defined as the number of physical blocks in the user memory of the particular RFID tag. For RFID tags with segmented memory structures, the number of blocks is required for each memory bank to which the application has access for a read or write commands.

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 Processor. The number of blocks is not communicated across the application interface.

9.2.4 AFI

The **AFI** is defined by the application as a single byte value and is used as an air interface selection mechanism to separate RFID tags with a given **AFI** from tags with different **AFI** values. The **AFI** is defined by the application as a single byte value compliant with ISO/IEC 15961 Part 3 and the register maintained as part of ISO/IEC 15961 Part 2.

9.2.5 DSFID

The **DSFID** is defined by the application as a single byte value with the following structure:

Bits 8 and 7	Determine the **Access-Method** (see Clause 11).
Bit 6	Is the extended syntax indicator bit (see 9.2.7).
Bits 5 to 1	Identify the **Data-Format**, as defined in ISO/IEC 15961 Part 3 and registered to ISO/IEC 15961 Part 2.

Once a **Data-Format** has been specified for an RFID tag (or a specific bank of a Segmented memory RFID tag), all subsequent additional data shall be encoded in a compliant manner. Once an **Access-Method** has been specified for an RFID tag (or a specific bank of a Segmented memory RFID tag), all subsequent additional data shall be encoded in a compliant manner, with one exception. It is possible to convert a **No-Directory** structure to a **Directory** structure. These points are discussed in greater detail for the specific **Access-Methods** (see Clause 11 and associated annexes).

The only alternative process to change the **Data-Format** and / or **Access-Method** is to invoke an erase command and then overwrite all the new data. This process might fundamentally change the original function of the application data on the RFID tag.

The single byte **DSFID** can support up to four **Access-Methods** and up to 30 **Data-Formats**. Extension mechanisms that can be applied independently to each are possible as defined in the rules of the following sub-clauses. These sub-clauses also define support for other functions that are not directly supported by air interface features.

9.2.6 Encoding the Extended-Data-Format

When the ISO/IEC 15961-1 specifies a **Data-Format** value greater than 31_{10}, the **Extended-Data-Format** byte is required to encode the value of the **Data-Format**. Bits 5 to 1 of the single (primary) **DSFID** are set to the value 11111 to indicate that an **Extended-Data-Format** byte follows. The value of the **Extended-Data-Format** byte is the value of the **Data-Format** provided by the application command less 32_{10}. For example Data-Format 69_{10} requires the **Extended-Data-Format** byte to have the value 00100101_2.

9.2.7 Other extensions using the Extended Syntax Indicator bit

If the Extended Syntax Indicator bit = 0, then all the information about the **Access-Method** is encoded in the single byte **DSFID** and no other functions are indicated as present. If the Extended Syntax Indicator bit = 1, then an Extended Syntax Flag Byte 1 is used (see 9.2.8).

9.2.8 Extended Syntax Flag Byte 1

The Extended Syntax Flag Byte 1 is defined by the application and/or by processes of the Data Processor, as will be described in various sub-clauses. The initial structure is as a single byte value with the following structure:

Bit 8	Extended Syntax Flag Byte 2 indicator (see 9.2.13)
Bits 7 & 6	Extensions to Access-Method (see Clause 11)
Bits 5 & 4	Memory length indicator (see 9.2.9)
Bits 3 & 2	Data CRC indicator (see 9.2.11)
Bit 1	Reserved

9.2.9 Memory length indicator bits

The memory length indicator bits are determined by arguments in an application command that recommends its use where the relevant memory is greater than 256 bits, taking into account the information provided by the air interface (see 9.2.9.1). There are a number of conditions that apply to particular **Access-Methods** that are discussed in 9.2.9.2. The 2-bit code values for the memory length indicator are:

00 No encoding of the total length of data or memory capacity, as these are small (i.e. not more than 256 bits)
01 Memory capacity is declared
10 The total length of encoded data is declared
11 Both memory capacity and total length of encoded data are declared

Irrespective of the capability of the air interface to provide a hardware solution for encoding the total length of encoded data, if the indicator bits are set, then the Data Processor shall update the encoded length each time that data is written to the RFID tag.

9.2.9.1 Air interface exceptions

The following air interface protocol shall make use of memory length encoding, as qualified by the **Access-Method** as follows:

— **ISO/IEC 18000-3 Mode 1:** The memory capacity does not need to be encoded, because this is provided by other means in the services from the air interface, particularly through part of the code value of the **Singulation-Id**. The length of the encoded data over 256 bits should be encoded, unless this is suppoerted by the **Access-Method**. The length shall be expressed in blocks, which can vary between compliant RFID tags.

— **ISO/IEC 18000-6 Type C:** Length encoding shall not be used either for the memory capacity or the encoded length for Memory Banks 00_2, 01_2, and 10_2. The total length of the encoded data over 256 bits in Memory Bank 11_2 should be encoded unless this is supported by the **Access-Method**. The

length shall be expressed in blocks as defined in this International Standard and expressed as 16-bit words in this air interface protocol.

— **ISO/IEC 18000-6 Type D:** The memory capacity does not need to be encoded because this is declared by the number of available pages of memory. The total length of encoded data is not required for the UII segment and the Item-related segment because these lengths are encoded on the tag using hardware processes.

9.2.9.2 Access-Method requirements and exceptions

Encoding the memory capacity and the total length of encoded data is applied to the **Access-Methods** in the following way:

No-Directory
Unless the Tag Driver has another means of declaring the memory capacity, this should be declared and encoded if this is greater than 256 bits. The total length of the encoded data should also be declared and encoded if this is greater than 256 bits.

Directory
The memory capacity should be encoded if greater than 256 bits, unless the Tag Driver has another means of declaring the memory capacity. The total length of the encoded data should apply to the **Directory** only, irrespective of its length.

If the **Directory** is added at some later stage over an existing **No-Directory** structure the original length of data encoding will not only be different but will refer to different bytes. The length of encoded Directory shall be overwritten at the time that the Directory is first prepared.

Packed Objects
Unless the Tag Driver has another means of declaring the memory capacity, this should be declared and encoded if this is greater than 256 bits. As each Packed Object declares its length, if only one Packed Object is encoded there is mo need to encode the total length of encoded data. The total length of the encoded data should be declared and encoded if there are two or more Packed Objects and the total length of all the encoded data is greater than 256 bits.

Tag Data Profile
The length of the encoding is declared in the encoded header record, so there is no requirement to encode length.

Multiple-Records
The memory capacity is encoded for each **Multiple-Record** in its preamble.

9.2.10 Procedure for length encoding

The procedure for encoding the memory capacity or total length of encoded data shall be based on the length encoding rules defined in Annex D.2. Table 5 — Examples of length encoding shows examples of length encoding.

Table 5 — Examples of length encoding

	0	0	0000000				
	1	0	0000001				
$2^7 - 1$	**127**	0	1111111				
2^7	**128**	1	0000001	0	0000000		
$2^{14} - 1$	**16383**	1	1111111	0	1111111		
2^{14}	**16384**	1	0000001	1	0000000	0	0000000
$2^{21} - 1$	**2097151**	1	1111111	1	1111111	0	1111111

9.2.11 Data CRC indicators

The data CRC indicator bits are generated by the Data Processor, as defined in this sub-clause. The use of a data CRC is determined by arguments in an application command.

The 2-bit code values for the data CRC indicator are:

00 No data CRC
01 Data CRC applied to each individual data set
10 Data CRC applied only to the entire encoded data
11 Data CRC applied to each data set and to the entire encoded data

9.2.12 Data CRC

The data CRC is calculated using the polynomial defined in ISO/IEC 13239 (which is also known as the CRC-CCITT International Standard, ITU Recommendation X.25): $x^{16} + x^{12} + x^5 + 1$. The 16-bit register shall be preloaded with $FFFF_{16}$ prior to calculating the CRC-16.

9.2.12.1 The Data CRC applied to a data set

The CRC-16 shall be applied to the entire data set, and encoded immediately after the data set. If, there is a requirement to block align after the CRC-16, then padding compliant with the Access Method shall be used. The CRC-16 shall be applied to all the data sets encoded on the RFID tag.

9.2.12.2 The Data CRC applied to the entire encoded bytes

The CRC-16 shall be applied to the entire encoded data plus the terminator byte (00_{16}) and shall be encoded immediately after the terminator byte.

9.2.13 Extended Syntax Flag Byte 2

The Extended Syntax Flag Byte 2 is defined by the application and/or by processes of the Data Processor, as will be described in various sub-clauses. The initial structure has the following structure:

Bit 8	Extended Syntax Flag Byte 3 indicator - an extension for future special features
Bit 7	Simple sensor indicator (see 9.2.14)
Bit 6	Battery-Assist indicator (see 9.2.15)
Bit 5	Full-function sensor indicator (see 9.2.16)
Bit s 4 to 1	Reserved

9.2.14 Simple Sensor indicator

Simple Sensors have limited configuration capabilities and can be factory programmed, so provide resultant observations based on their pre-designed functionality. The Simple Sensor indicator is used when one such sensor is added to an RFID tag. The indicator is only used where the functionality, if supported on the RFID tag, is not declared by some air interface mechanism. The indicator is set to “1” if the RFID tag supports a Simple Sensor.

The information is transmitted in any response that contains the Extended Syntax Flag Byte, but any follow-up action by the interrogator or the application is beyond the scope of this International Standard.

9.2.15 Battery Assist indicator

The Battery Assist indicator is used when a battery is added to a passive tag to assist with improving the communication capability of the RFID tag. The indicator is only used where the functionality, if supported on the RFID tag, is not declared by some air interface mechanism. The indicator is set to “1” if the RFID tag employs battery-assist power.

The information is transmitted in any response that contains the Extended Syntax Flag Byte 2, but any follow-up action by the interrogator or the application is beyond the scope of this International Standard.

9.2.16 Full-Function Sensor indicator

The Full-Function Sensor indicator is used when one or more such sensors are added to an RFID tag. The indicator is only used where the functionality, if supported on the RFID tag, is not declared by some air interface mechanism. The indicator is set to “1” if the RFID tag supports one or more Full-Function Sensors.

The information is transmitted in any response that contains the Extended Syntax Flag Byte 2, but any follow-up action by the interrogator or the application is beyond the scope of this International Standard.

9.2.17 DSFID and Extended Syntax

This syntax depends on the value of the **Data-Format** and Extended Syntax Indicator bit. The following sequence of elements shall be used, with elements omitted if not signalled by earlier bytes:

NOTE The term "shall follow" by each element depends on how the DSFID is encoded, and whether all the preceding elements are declared to be present. All the elements shall precede any encoding of data compliant with the declared Access-Method.

DSFID: This shall be encoded according to the rules of the air interface protocol, which could be in a separate memory location or with other user-related data.

Extended-Data-Format: If bits 5 to 1 have the value 11111, then this shall follow the **DSFID**.

Extended Syntax Flag Byte 1: If the **Extended Syntax Flag Byte 2 Indicator** bit is set in the **Extended Syntax Flag Byte 1**, then this byte shall be the next logical element. In turn, it determines which of the following optional elements are encoded, which depend on the air interface and **Access-Method** constraints as discussed above.

Extended Syntax Flag Byte 2: If the **Extended Syntax Flag Byte 3 Indicator** bit is set in the **Extended Syntax Flag Byte 2**, then this byte shall be the next logical element. In turn, it determines which of the following optional elements are encoded, which depend on the air interface and **Access-Method** constraints as discussed above.

Syntax Extension (provisional): If future editions of this International Standard specify and support this element, then it will appear in this logical sequence in the structure.

Memory Capacity: This is a single or multiple bytes indicating the size of the memory in terms of the block size definition provided by the Tag Driver for the particular tag. It is not required if the memory capacity is 256 bits or less.

Total Encoded Data Length: This is a single or multiple bytes indicating the length of the encoded bytes in terms of the block size definition provided by the Tag Driver for the particular tag. It is not required if the total length of encoded data is 256 bits or less.

Pad Byte(s): If there is a requirement to lock this extended system information, or for example to provide memory for length encoding to be added at a later time, it might be necessary to align to a block boundary. The **Null-Byte** mechanism, encoding a sequence of byte 80_{16} shall be used (see Annex D.6.4).

Table 6 — DSFID extension syntax shows all the syntax structures that are possible, including how future extensions will be addressed. The table only shows bytes that are encoded under the different conditions, and shows these in the sequence in which they would appear to a decoder. Bytes that signal a feature that is invoked beyond the extended DSFID (e.g. a data CRC) are shown with 'x' in the relevant bit position. The presence of these is only relevant to the byte in which they are encoded and does not affect the structure of the remaining bytes in the Extended DSFID.

Table 6 — DSFID extension syntax

DSFID	Extended-Data-Format	Extended Syntax Flag Byte 1	Extended Syntax Flag Byte 2	Syntax Extension	Memory Capacity	Encoded Length	Pad Bytes
bb000000 to bb011110	Not present	Not present	Not present	Reserved	Not present	Not present	Not present
bb011111	Present	Not present	Not present	Reserved	Not present	Not present	Conditional
bb100000 to bb111110	Not present	0bb00000 Access-Method only	Not present	Reserved	Not present	Not present	Conditional
bb100000 to bb111110	Not present	0xx00xxx	Not present	Reserved	Not present	Not present	Conditional
bb100000 to bb111110	Not present	0xx01xxx	Not present	Reserved	Present	Not present	Conditional
bb100000 to bb111110	Not present	0xx10xxx	Not present	Reserved	Not present	Present	Conditional
bb100000 to bb111110	Not present	0xx11xxx	Not present	Reserved	Present	Present	Conditional
bb100000 to bb111110	Not present	1xx00xxx	Present	Reserved	Not present	Not present	Conditional
bb100000 to bb111110	Not present	1xx01xxx	Present	Reserved	Present	Not present	Conditional
bb100000 to bb111110	Not present	1xx10xxx	Present	Reserved	Not present	Present	Conditional
bb100000 to bb111110	Not present	1xx11xxx	Present	Reserved	Present	Present	Conditional
bb111111	Present	0bb00000 Access-Method only	Not present	Reserved	Not present	Not present	Conditional
bb111111	Present	0xx00xxx	Not present	Reserved	Not present	Not present	Conditional
bb111111	Present	0xx01xxx	Not present	Reserved	Present	Not present	Conditional
bb111111	Present	0xx10xxx	Not present	Reserved	Not present	Present	Conditional
bb111111	Present	0xx11xxx	Not present	Reserved	Present	Present	Conditional
bb111111	Present	1xx00xxx	Present	Reserved	Not present	Not present	Conditional
bb111111	Present	1xx01xxx	Present	Reserved	Present	Not present	Conditional
bb111111	Present	1xx10xxx	Present	Reserved	Not present	Present	Conditional
bb111111	Present	1xx11xxx	Present	Reserved	Present	Present	Conditional

NOTE: In the table, b indicates any value
x indicates that the value is not relevant to the subsequent structure

The "conditional" state for Pad bytes depends on the air interface support for selective locking, the call by the application to lock this syntax sequence, or alternatively to leave it unlocked but to lock the encoded data.

The following hypothetical example illustrates various aspects of encoding the DSFID and other features. The data to be encoded is as follows:

Access Method = 5
Data format = 69
Memory capacity = 128 blocks
Length of encoded data = 37 blocks

Data CRC applied to all the data
A simple sensor is on the tag
Any pad bytes to enable the length of the encode data to be the same size as the memory capacity.

The Data Format = 69 is encoded to the rules defined in 9.2.6. The calculation determines the value of the byte that immediately follows the DSFID byte:

$$69 - 32 = 37 = 00100101_2$$

The Access Method = 5 is encoded to rules defined in

Table 8 — Assigned and reserved Access-Methods. Four bits need to be encoded 0010_2 spread over different bytes. The lead bits of the DSFID are encoded with 00_2 and the next bit is set = 1_2 to signal that the Extended Syntax Flag Byte 1 needs to be encoded. The value of the first two bytes can be fully determined and contain the DSFID byte and Extended data format byte:

$$00111111_2 \quad 00100101_2..= \quad 3F25_{16}$$

The Extended Syntax Flag Byte 1 has the following initial structure to encode the other two bits of the Access Method (see 9.2.8):

$$x10xxxxx_2$$

Encoding the indicators for memory capacity and length of the encoded data determines two more bits:

$$x1011xxx_2$$

Encoding that a Data CRC is applied to the entire data but not to the individual data sets determines two more bits:

$$x101110x_2$$

The final bit is reserved, so has the default value 0_2. As this tag has a simple sensor that is not declared by an air interface mechanism, the Extended Syntax Flag Byte 2 is required, so the lead bit of this flag needs to be set = 1_2. The complete encoding of the Extended Syntax Flag Byte 1 is:

$$11011100_2 \; = \; DC_{16}$$

This byte follows immediately after the encoded bytes that have already been determined, resulting in the following string.

$$3F25DC_{16}$$

The Extended Syntax Flag Byte 2 has the following initial structure to encode the fact that there is a simple sensor encoded on the RFID tag (see 9.2.13):

$$x1xxxxxx_2$$

There is no indicator for battery assist and full function sensors, so the structure is set as follows:

$$x100xxxx_2$$

As there are no further extensions defined, the lead bit = 0_2 and the trailing bits are = 0000_2 to indicate that they are reserved, resulting in this encoding:

$$01000000_2 \; = \; 40_{16}$$

The byte string is now extended to:

$$3F25DC40_{16}$$

The encoding for the memory capacity follows the rules defined in 9.2.10, resulting in these two bytes

$$10000001_2 \qquad 00000000_2 = 8100_{16}$$

The length of the encoded data is 37 blocks, encoded as 25_{16}. These three bytes are appended to the four already determined:

$$3F25DC40810025_{16}$$

Finally as the memory capacity requires two bytes but the length of the current encoded data only requires one byte, it is necessary to provide a single pad byte, as described in 9.2.17. The complete encoded DSFID byte string comprises: $3F25DC4081002580_{16}$.

9.3 Configuring the Logical Memory

The **Logical Memory** is assumed to be empty at the beginning of the process of any application read or write command. This enables the Data Processor to populate the Logical Memory with encoded bytes using the encoding rules of this International Standard. It also enables transfers across the air interface (usually in blocks) to partially populate the **Logical Memory**, to enable decoding by the rules of this International Standard.

The size of the **Logical Memory** is defined using the physical block size and number of blocks from the system information (see 9.2.2 and 9.2.3). 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. However other structures such as arrays and byte streams are also likely to be used in implementations.

The **Access-Method** determines additional structuring rules (see Clause 11 for a detailed definition).

10 The Command/Response Unit: processing of command and response arguments

The Command/Response Unit is responsible for processing application commands and providing responses to the application system from the RFID tag(s) with which it is communicating. This unit can be considered conceptually as some form of executive software that receives application commands as defined in ISO/IEC 15961-1 and illustrated in Figure 3 — High level information flows. It takes account of the support for application commands by the particular air interface protocol; this information being delivered by the Tag Driver. It then calls for the rules for encoding (or decoding) and formatting functions as defined by the **Access-Method**.

Additionally, the command arguments concerned with processing (e.g. lock functions, conditional selections and so on) call for interfaces with the interrogator to transfer this as particular ISO/IEC 18000 air interface command sets. Throughout this process, the Command/Response unit is continually monitoring and updating the Logical Memory of each RFID tag with which it is maintaining a logical association.

The arguments used in the application commands are classified as:

— process arguments, which determine that some action is required by the Data Processor or the interrogator to achieve successful encoding or decoding of the bytes on the RFID tag. These are defined in 10.1.

— **Completion-Codes**, which are command-related error or pass codes. These are described in 10.2.

— **Execution-Codes**, which are system-related error or pass codes. These are described in 10.3.

— command-related and response-related field names, which distinguish a particular type of data, and for this International Standard simply act as descriptors for data which is input to the encode process or is output from the decode process. These are listed below and defined in ISO/IEC 15961-1.

Data-Set
Identities
Lock-Status
Logical-Memory-Map
Number-of-Tags-Found
Protocol-Control-Word
Read-Data

10.1 Process arguments

The following sub-clauses define the process arguments.

10.1.1 Access-Password

The **Access-Password** is a 32-bit (4 byte) code that provides permission to access data on the RFID tag either to read or to write. Of the Tag Drivers currently supported, the **Access-Password** argument only applies to ISO/IEC 18000-6C tags.

The **Access-Password** is included as an argument in the following ISO/IEC 15961-1 application commands: **Write-Objects-Segmented-Memory-Tag**, **Write-EPC-UII**, and **Read-Words-Segmented-Memory-Tag**. If the **Access-Password** is a non-zero value, then this 32-bit code needs to match with that on the RFID tag to gain access for subsequent secure transactions. The interrogator processes this according to the rules of the air interface protocol, with the password separated into two 16-bit strings and processed according to the rules of ISO/IEC 18000-6C. If a perfect match is achieved of the 32-bit code, then the other arguments in the application commands can be processed.

ISO/IEC 18000-6C indicates that the **Access-Password** itself is optional, as is support for the *Access* air interface command both on interrogators and RFID tags. If any of these optional features are not present, then the **Completion-Code** (25) **Password-Mismatch** is returned.

10.1.2 Additional-App-Bits

The **Inventory-ISO-UIImemory** command enables a search on an ISO/IEC 18000-6C tag to match a bit pattern on the command with that on one or more RFID tags in the field. The application command supports selection on the **AFI** and **DSFID**. The **Additional-App-Bits** field allows the bit pattern to be extended to enable a search to a finer resolution. For example, it could include the value of the Precursor and the **Relative-OID**. If this field is included in an application command, it is concatenated to the **AFI** and **DSFID** and incorporated directly into the associated ISO/IEC 18000-6C *Inventory* air interface command.

This argument is used with ISO/IEC 18000-6C when accessing **Object-Identifier** related data.

10.1.3 AFI

The application command argument **AFI** is used as parameter in the construction of an air interface command. Processing this argument basically maps the value of the **AFI** into bit positions in the air interface command. The **AFI** might not be explicitly called out in the command, but its location within the encoded memory is specified in the relevant ISO/IEC 18000 standard.

This argument is used with ISO/IEC 18000-6C.

10.1.4 AFI-Lock

The application command argument **AFI-Lock** is BOOLEAN. If set to TRUE the **AFI** shall be written to the RFID tag's system information and be permanently locked. If set to FALSE it shall be written but not locked.

The **AFI-Lock** argument is part of the **Configure-AFI** command, which is not directly supported by ISO/IEC 18000-6C (see Annex A.3.1).

Possible failures to complete the command argument result in the following error, represented by the **Completion-Codes** (more fully defined in 10.2):

1 AFI-Not-Configured
2 AFI-Not-Configured-Locked
3 AFI-Configured-Lock-Failed

10.1.5 Append-To-Existing-Multiple-Record

The application command argument **Append-To-Existing-Multiple-Record** is a BOOLEAN argument. If TRUE, the data objects are to be appended to an existing multiple record, subject to various checks by the Data Processor. If FALSE, then a new multiple record shall be created.

10.1.6 Application-Defined-Record-Capacity

The application command argument **Application-Defined-Record-Capacity** is a BOOLEAN argument that applies to a multiple record. If FALSE, the Data Processor automatically determines that that the capacity for the given multiple record is simply based on the encoded data and then increased to be block aligned. If set to TRUE, then the application sets the size of memory assigned to the record using the **Record-Memory-Capacity** argument. This is the value used by the Data Processor.

10.1.7 Avoid-Duplicate

The application command argument **Avoid-Duplicate** is used in a write command to ensure that the **Object-Identifier** is not already encoded on the RFID tag.

This is a BOOLEAN argument. If set to TRUE, sufficient content of the memory is read into the Logical Memory until a duplicate **Object-Identifier** is found, or until the end of the encoding. Variant processes based on the **Access-Methods** are as follows:

No-Directory: sufficient content of each **Data-Set** shall be read to identify the **Object-Identifier**.

Directory: the directory can be read to establish the presence of the candidate **Object-Identifier**.

Packed-Objects: the index of each **Packed-Object** is read to establish whether the candidate **Object-Identifier** is present.

Tag-Data-Profile: sufficient content of each **Data-Set** shall be read to identify the **Object-Identifier**.

Multiple-Records: The specific **Object-Identifier** structure (see 11.5.2) identifies a data element within a specific record. The final arc identifies the data element, which shall be used to apply this argument within the record using the processes (above) for the **Access-Method** of the record.

If a duplicate of the specified **Object-Identifier** is found, the write process is aborted and the appropriate **Completion-Code** is used to indicate the error.

If the BOOLEAN argument is 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 **Completion-Codes**:

9 Object-Not-Added
10 Duplicate-Object

10.1.8 Check-Duplicate

The application command argument **Check-Duplicate** is used in a read command to ensure that the **Object-Identifier** is not already encoded on the RFID tag. This is basically a check of encoded data that is either non-compliant with this International Standard, or where the **Avoid-Duplicate** argument was set to FALSE in the original write command, possibly by a trading partner.

This is a BOOLEAN argument. If set to TRUE, sufficient content of the memory is read into the logical memory until a duplicate **Object-Identifier** is found, or until the end of the encoding. The variant processes based on the **Access-Method** are as defined in 10.1.7 for the **Avoid-Duplicate** argument. If a duplicate of the specified **Object-Identifier** is found, the **Completion-Code** (10) is used to indicate the error, and neither **Object-Identifier** nor associated **Object** is returned as a response, because there is no way to ascertain which is valid. Other housekeeping commands such as **Read-Logical-Memory-Map** can then be used to identify errors on the tag.

If the BOOLEAN argument is 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 **Completion-Codes**:

10 Duplicate-Object
13 Object-Identifier-Not-Found

10.1.9 Compact-Parameter

The application command argument **Compact-Parameter** determines what compaction scheme is applied to the data **Object**, based on the following integer code values:

0 **Application-Defined** - The **Object** shall not be processed through the data compaction rules of ISO/IEC 15962 and remains unaltered when stored in the Logical Memory Map of the RFID tag. This compaction applies to the **No-Directory** and **Directory Access-Methods**.

1 **Compact** - This requires using the basic ISO/IEC 15962 compaction rules to compact the **Object** as efficiently as possible to reduce the number of bytes required on the Logical Memory Map. This compaction applies to the **No-Directory** and **Directory Access-Methods**.

2 **UTF8-Data** - This identifies that the **Object** has been externally transformed from the ISO/IEC 10646 coded character set to UTF-8 encoding. The **Object** shall not be processed through the data compaction rules of ISO/IEC 15962 and remains unaltered for transfer to the Logical Memory Map. This compaction applies to the **No-Directory** and **Directory Access-Methods**.

3 **Pack-Objects** - This identifies that a set of **Objects** is to be encoded using the **Packed-Object** encoding scheme and identified with the associated **Access-Method**. If any of the set is defined with a different **Compact-Parameter** an error occurred in the command and encoding shall be aborted. The set of **Objects** shall be compacted using the **Packed-Object** encoding rules.

4 **Tag-Data-Profile** - This identifies that a set of Objects is to be encoded using the Tag Data Profile encoding scheme and identified with the associated **Access-Method**. Although the set of compaction schemes is identical to the basic ISO/IEC 15962 compaction rules, the Profile IDTable can call for a specific compaction scheme.

5 to 14 reserved for future definition.

15 **De-Compacted-Data** – This identifies that the **Object** in a response has been de-compacted using rules in ISO/IEC 15962 and restored to its original application input format. This code applies to any de-compacted data from **Access-Methods**: **No-Directory**, **Directory**, and **Packed-Objects**. It does not apply to **Objects** that were originally encoded as **Application-**

Defined or **UTF8-Data**, because these codes are carried throughout the command arguments encoded on the tag, and response to the application.

The **Packed-Object** encoding scheme may only be applied if the application administrators choose to adopt the scheme by creating an index table to which source references should be provided in the register of data constructs (associated with ISO/IEC 15961-2).

10.1.10 Data-Length-Of-Record

This application response argument provides the size of the encoded Multiple Record in terms of the write block size, as encoded in EBV-8 format in the record preamble. The Data Processor returns this in EBV-8 format.

10.1.11 DSFID

The application command argument **DSFID** is used as parameter in the construction of an air interface command. Processing this argument basically maps the value of the **DSFID** into bit positions in the air interface command. The **DSFID** might not be explicitly called out in the command, but its location within the encoded memory is specified in the relevant ISO/IEC 18000 standard.

This argument is used with ISO/IEC 18000-6 Types C and D.

10.1.12 DSFID-Lock

The application command argument **DSFID-Lock** is BOOLEAN. If set to TRUE the **DSFID** shall be written to the RFID tag's system information and be permanently locked. If set to FALSE it shall be written but not locked.

The **DSFID-Lock** argument is part of the **Configure-DSFID** command, which is not directly supported by ISO/IEC 18000-6C (see Annex A.3.2).

Possible failures to complete the command argument result in the following error, represented by the **Completion-Codes** (more fully defined in 10.2):

4 DSFID-Not-Configured
5 DSFID-Not-Configured-Locked
6 DSFID-Configured-Lock-Failed

10.1.13 Directory-Length-EBV8-Indicator

This application command argument is used to determine the size of the EBV-8 field in the MR-header that encodes the size of the directory. The following codes apply:

1 Single byte EBV-8
2 Double byte EBV-8, even if the length is less than 128 blocks

The value 2 is used for a directory that might originally be small in size, but is expected to increase in size at some future time. The Data Processor shall encode the EBV-8 using the length defined by the application command.

10.1.14 Encoded-Memory-Capacity

This application response argument provides the size reserved for a Multiple Record in terms of the write block size, as encoded in EBV-8 format in the record preamble. The Data Processor returns this in EBV-8 format.

10.1.15 EPC-Code

The application command argument **EPC-Code** provides a unique code defined by EPCglobal Tag Data Standard. Because some of the codes do not align on an 8-bit boundary, these need to be rounded to 8-bit bytes for processing through the Data Processor, and be rounded to 16-bit words for encoding on the associated RFID tag. No other processing is required.

10.1.16 Hierarchical-Identifier-Arc

The application response argument **Hierarchical-Identifier-Arc** is an integer value, converted by the Data Processor from the EBV-8 value encoded in the Multiple Records preamble from a hierarchical record.

10.1.17 Identifier-Of-My-Parent

The command and response argument **Identifier-Of-My-Parent** is used for a multiple record that is part of a hierarchy, and has the value of the hierarchical code of that record. The Data Processor shall check that the associated parent record exists by establishing that the hierarchical code exists. In addition it shall check that it can be a parent from bits 8 to 6 of the Multiple-Records intermediate arcs indicator having the values 010 or 011. If the hierarchical code exists and the fact that the record can be a parent is proven then the record can be written. Otherwise the encoding process is discontinued and an error reported.

10.1.18 Identify-Method and Number-Of-Tags

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

0 **Inventory-All-Tags** - This calls for the **Singulation-Id** of all RFID tags in the operating field to be identified, probably with the additional system constraint of a time limit.

1 **Inventory-At-Least** - This calls for the **Singulation-Id** of all RFID 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 **Inventory-No-More-Than** - This calls for the **Singulation-Id** of a maximum number of RFID tags in the operating field to be identified. This could be used for sampling purposes.

3 **Inventory-Exactly** - This calls for the **Singulation-Id** of a specific number of RFID 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 **Number-Of-Tags** is a conditional qualifier to the basis argument **identify-Method**. This numeric selection is defined by the application to confirm by an actual count, or a partial count, of which particular RFID tags are in the operating field. The Data 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 **Identify-Method** is set to **Inventory-All-Tags (0)**, the **Number-Of-Tags** should be set to 0 in the application argument.

— If the **Identify-Method** is set to **Inventory-At-Least (1)** and the **Number-Of-Tags** is set to 1, the RFID interrogator is effectively being set to wait for an RFID tag to enter the operating field. It then responds with the **Singulation-Id** of the first RFID tag that it detects, and others that are in the

operating area at the same time. If the **Number-Of-Tags** is set to greater than 1, the argument might need to supported with an additional system constraint of a time limit.

— 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 **Completion-Code**.

10.1.19 Instance-Of-Arc

The application response argument **Instance-Of-Arc** is an integer value, converted by the data Processor from the EBV-8 value encoded in the Multiple Records preamble.

10.1.20 Item-Related-DSFID

The application argument **Item-Related-DSFID** is used as parameter to identify the DSFID encoded with the Item-Related segment of an ISO/IEC 18000-6 Type D tag.

10.1.21 Item-Related-Segment-Map

The application response argument **Item-Related-Segment-Map** is used as parameter to identify the raw byte string encoded within the Item-Related segment of an ISO/IEC 18000-6 Type D tag.

10.1.22 Kill-Password

The application command argument **Kill-Password** requires the Data Processor to pass this to the interrogator so that the **Kill-Password** in the command is matched with that on the RFID tag. A match results in the RFID tag function being permanently disabled. A mismatch results in the air interface rejecting the command.

10.1.23 Length-Of-Mask

The application command argument **Length-Of-Mask** represents the number of bits in the **Tag-Mask** and is used as a parameter in the associated ISO/IEC 18000-6C *Inventory* air interface command.

This argument is used with ISO/IEC 18000-6C when accessing EPCglobal related data.

10.1.24 Lock-Directory-Entry

The application command argument **Lock-Directory-Entry** is BOOLEAN and if set to TRUE the interrogator shall lock the directory entry for the particular record.

10.1.25 Lock-Multiple-Records-Header

The application command argument **Lock-Multiple-Records-Header** is used to determine if all, some, or none of the MR-header is locked. This command argument is presented as an integer value and the following codes apply:

0 Not locked
1 Completely locked
2 The Number of Records field remains unlocked, the preceding fields are locked.
3 The Data Length of the Directory field remains unlocked, all other fields including the Number of Records field are locked.
4 The Data length of the directory field and the Number of Records field remain unlocked, all other fields are locked.

The Data Processor shall ensure block alignment. The interrogator shall lock the memory as instructed by the command.

10.1.26 Lock-Record-Preamble

The application command argument **Lock-Record-Preamble** is BOOLEAN and if set to TRUE the interrogator shall lock the fields in the preamble.

The Data Processor shall ensure block alignment. The interrogator shall lock the memory as instructed by the command.

10.1.27 Lock-UII-Segment-Arguments

The application command argument **Lock-UII-Segment-Arguments** defines which componemt parts of the UII segment of an ISO/IEC 18000-6 Type D tag to lock. The size of the lock block is determined by the tag model declared by the TID-S page(s) of the tag.

10.1.28 Max-App-Length

The application command argument **Max-App-Length** is provided as a numeric value equivalent to the number of bytes to be read starting from the first location on the RFID tag. This has to be converted into blocks, by dividing the value of the **Max-App-Length** by the block size, and rounding up to a whole block as necessary.

The number of blocks is then transferred to the interrogator for the air interface protocol to read and return this number of blocks, which should provide the application with the first **Data-Set(s)** encoded on the RFID tag.

The **Max-App-Length** argument is only applicable to the **Access-Method No-Directory** and **Directory**. It should be applicable to all single bank memory structures, and to Memory Bank 11_2 of the ISO/IEC 18000-6C RFID tag and other segmented memory structures.

10.1.29 Memory-Bank

The application command argument **Memory-Bank** represents a 2-bit code used to distinguish which memory bank needs to be accessed to invoke the following commands:

— **Read-Words-Segmented-Memory-Tag**, with possible values (01_2 to 11_2)

— **Write-Objects-Segmented-Memory-Tag**, with possible values (01_2 or 11_2)

These bit values are incorporated directly into the associated ISO/IEC 18000-6C air interface commands.

10.1.30 Memory-Bank-Lock

The application command argument **Memory-Bank-Lock** is BOOLEAN. If set to TRUE, the selected memory of the ISO/IEC 18000-6C tag shall be permanently locked. If set to FALSE, any data written to the tag shall not be locked in the specified memory bank.

The **Memory-Bank-Lock** argument is part of the **Write-EPC-UII** command, which is only supported by ISO/IEC 18000-6C.

NOTE For the **Write-ISO-UII** and other ISO-related data to be encoded in an ISO/IEC 18000-6C tag, different rules apply.

10.1.31 Memory-Segment

The application command argument **Memory-Segment** is used to distinguish which memory segments of an ISO/IEC 18000-6 Type D are addressed.

10.1.32 Memory-Type

The command argument **Memory-Type** is used to define which memory structure is intended for encoding a **Monomorphic-UII**, effectively instructing the Data Processor which of the optional and conditional arguments and processes to apply to the encoding process.

10.1.33 Multiple-Records-Directory-Length

The application response argument **Multiple-Records-Directory-Length** is an EBV-8 value. If the data length of directory is not encoded in the MR-header, then a zero value is returned.

10.1.34 Multiple-Records-Features-Indicator

The application command and response argument **Multiple-Records-Features-Indicator** is a bit map. The 8-bit value in the command shall be encoded as submitted in the MR-Header. As currently defined in Annex Q.2.7 some of the values for bits 7 to 4 are either not permitted or are reserved. Bit 1 is also currently reserved.

The Data Processor shall abort processing the **Configure-Multiple-Records-Header** command if any of the bits that are not permitted or are reserved are included in the application command for this argument.

10.1.35 NSI-Bits

The application command argument **NSI-Bits** defines a 9-bit code defined by EPCglobal used as a prefix to the **EPC-Code** when encoded on the associated RFID tag. The NSI-Bits are encoded in bit locations 17h to 1Fh of memory bank 01_2 of an ISO/IEC 18000-6C tag, only when encoding EPCglobal data.

10.1.36 Number-In-Data-Element-List

The application command argument **Number-In-Data-Element-List** is only used in a hierarchical record that is a data element list. The value is the count of the number of instances-of the data element being encoded. The Data Processor should use this as an aid to ensure that the number of data elements is matches this value. A mis-match is not treated as an error, so the encoding process continues.

10.1.37 Number-Of-Records

The response argument **Number-Of-Records** is used in one of two ways. If bit 2 of the **Multiple-Records-Features-Indicator** equals:

— 1 then this field is fully maintained as new records are added. A zero value is encoded when the MR-header is created. In this case the Data Processor returns the EBV-8 value as encoded in the MR-header.

— 0 then this field is not maintained and the value should be zero, but any other value should be ignored, for example if some record count was initially maintained but later this function was decided to be stopped. In this case the Data Processor returns the EBV-8 value of zero, irrespective of the value encoded in the MR-header.

10.1.38 Object-Lock

The application command argument **Object-Lock** is BOOLEAN. If set to TRUE, the selected **Object** and **Object-Identifier** (as presented in the ISO/IEC 15961-1 commands) shall be locked according to the rules that apply for the Tag Drivers. If set to FALSE, neither the **Object**, nor its associated **Object-Identifier**, shall be locked.

The sequence of bytes that need to be locked will vary according to the **Access-Method**, and further discussions about locking **Objects** and associated **Object-Identifiers** is included in Clause 11.

Additional rules apply to ISO/IEC 18000-6C tags (see Annex A.3.7) for details.

10.1.39 Packed-Object-Directory-Type

The application command argument **Packed-Object-Directory-Type** is used in write and modify commands to either indicate whether the particular Pack Object is treated as a Directory Packed Object or to change the status to be a directory. The following codes apply:

0	This **Packed-Object** is not a directory and does not require one
1	**Packed-Object** Presence/Absence
2	**Packed-Object** index field
3	**Packed-Object** offset
4	**Packed-Object** pointer-allocation only expecting a future command to set the directory type

Code '0' is used in a write command to identify that the Packed Object is not a Directory Packed Object. Code '5' is used to indicate that a directory will be set at some future time. The other three codes define different types of directory.

10.1.40 Password

The application command argument **Password** defines the byte string that represents a code value, qualified by the **Password-Type**, in a command that writes the password to the RFID tag. The **Password-Type** determines the structure of the **Password**, and if wrong the command shall be rejected. The **Password-Type** also determines the location to which the **Password** is written for the particular air interface protocol.

10.1.41 Password-Type

The application command argument **Password-Type** determines which location a password is written to in the specified memory bank of an RFID tag with a segmented memory. This argument qualifies the type of **Password** with the following code values.

0	Kill-Password
1	Access-Password

Processing this argument requires the password defined by the **Password-Type** to be specified to be written to the appropriate address location in the correct memory bank using a generic air interface *Write* command.

This argument is used with ISO/IEC 18000-6C.

10.1.42 Pointer

The application command argument **Pointer** identifies the hexadecimal address of the first (msb) bit against which to apply the **Tag-Mask** in the **Inventory-EPC-UIImemory** command. This field is incorporated directly into an associated ISO/IEC 18000-6C air interface command. It is also used in conjunction with **Length-Of-Mask.**

This argument is used with ISO/IEC 18000-6C when accessing EPCglobal related data.

10.1.43 Pointer-To-Multiple-Records-Directory

The application command and response argument **Pointer-To-Multiple-Records-Directory** is used to define the highest block number at the start address of the directory. This is defined as an EBV-8 value. If a directory is not initially encoded, then a fixed length EBV-8 string of the same length as required for the start point shall be encoded with a value zero. The start of the directory is not necessarily the highest address in Logical Memory, because other hardware features of the tag might be specified to be located there.

The Data Processor shall encode the value provided by the application command, unless the value is greater than the highest addressable memory location. In this case the command processing shall be aborted and the MR-header not encoded.

10.1.44 Read-Record-Type

The application command argument **Read-Record-Type** is used to identify various logical structures from the RFID tag. This command argument is presented as an integer value and the following codes apply:

0 **Read-Multiple-Records-Header**
1 **Read-Multiple-Records-Header-Plus-1st-Preamble**
2 **Read-Multiple-Records-Directory**
3 **Read-Preamble-Specific-Multiple-Record**
4 **Read-All-Record-OIDs-Specific-Record-Type**
5 **Read-OIDs-Specific-Multiple-Record**
6 **Read-All-Objects-Specific-Multiple-Record**
7 **Read-Multiple-Objects-Specific-Multiple-Record**
8 **Read-1st-Objects-Specific-Multiple-Record**
9 **Read-Data-Element-List-Specific-Multiple-Record**

The process rules are defined in 8.3.8.1.

10.1.45 Read-Type

The application command argument **Read-Type** defines which **Object-Identifier** is to read from the RFID tag. There are three **Read-Type** codes:

0 Read-First-Objects
1 Read-Multiple-Objects
2 Read-All-Objects

The **Read-First-Objects** argument is restricted to the **No-Directory** and **Directory Access-Methods**. If this **Read-Type** is selected, then the command needs to include the list of **Object-Identifiers** together with a value for the **Max-App-Length** (see 10.1.28).

The **Read-Multiple-Objects** argument requires a list of **Object-Identifiers** to be provided, and can be applied to any of the **Access-Methods**. If the **Check-Duplicate** argument is set to FALSE for all of the **Object-Identifiers**, the read process may cease once all the **Object-Identifiers** on the read list have been identified. If any of the **Object-Identifiers** have the **Check-Duplicate** argument set to TRUE, then reading continues to the end of the memory or until a duplicate **Object-Identifier** is found at which point the read process is aborted and an error is returned.

The **Read-All-Objects** argument does not require a list of **Object-Identifiers**. If any **Object-Identifier** is found to be duplicated, then the **Completion-Code** 10 should be returned and the reading process aborted. An appropriate “housekeeping” command should then be used to read the bytes from the RFID tag.

10.1.46 Record-Memory-Capacity

The application command argument **Record-Memory-Capacity** is used in commands to write a multiple record to indicate the amount of memory (in terms of write blocks) to assign, usually to enable the record to have additional data elements added. It is only used if the application needs to over-ride the automatic sizing by the Data Processor, when the **Application-Defined-Record-Capacity** argument is set to TRUE.

10.1.47 Record-Type-Arc

The application response argument **Record-Type-Arc** is an integer value, converted by the Data Processor from the EBv-8 value encoded in the Multiple Records preamble.

10.1.48 Record-Type-Classification

The application command and response argument **Record-Type-Classification** is a bit string that identifies the class of record being encoded. The code values are as defined in Annex Q.3.4.

The Data Processor shall check that the value for this argument in the command aligns with the **Object-identifier** in the command (particularly the value of the fourth arc and the value of the instance-of or hierarchical-id). For hierarchical records there also needs to be a logical agreement with the presence of a parent code. Any discrepancy points to an ambiguous error in the construction of the command, and the Data Processor shall ne proceed with the command.

10.1.49 Sector-Identifier

The **Sector-Identifier** is used in the MR-header either to indicate the true **Sector-Identifier** for all records in the Logical Memory, or to signal that the true value varies between records and the true value is encoded in the record. This application command and response argument is presented as an integer value. The Data Processor accepts the integer value provided in the command and converts this to EBV-8 format before encoding in the MR-header.

In commands to write, read, modify or delete, the **Sector-Identifier** is provided as part of the **Object-Identifier** and is already in EBV-8 format. The Data Processor needs to compare the value in the MR-header with the value in the command. If the MR-header value is:

0, then the command value shall be any value >2
1, then the command value shall be = 1
2, then the command value shall be = 2
>2. then the command value shall be the same

All other states are in error and the command shall not be invoked.

10.1.50 Segment-Read-Type

The application command argument **Segment-Read-Type** defines which segment, and to what detail, segments from an ISO/IEC 18000-6 type D are to be read. There are four **Segment-Read-Type** codes:

0	Read-Segments
1	Read-All-Segment-Details
2	Read-UII-Segment-Details
3	Read-Item-Related-Segment-Details

Read-Segments requires the Data Processor to parse the payload into a set of byte strings that represent: the TID segment, the UII segment, the Item-Related segment (if present), and the Simple Sensor Data Block (if present).

Read-UII-Segment-Details requires the Data Processor to provide in the response: the AFI, DSFID, Object Identifier, and data object.

Read-Item-Related-Segment-Details requires the Data Processor to provide in the response: the DSFID and list of object identifiers.

Read-All-Segment-Details requires the Data Processor to provide: the TID segment, the UII segment to the detail of the **Read-UII-Segment-Details** argument, the Item-Related segment to the detail of the **Read-Item-Related-Segment-Details** argument, and the Simple Sensor Data Block.

10.1.51 Simple-Sensor-Data-Block

The application response argument **Simple-Sensor-Data-Block** is used as parameter to identify the raw byte string encoded for simple sensors on an ISO/IEC 18000-6 Type D tag.

10.1.52 Start-Address-Of-Record

This application response argument provides the address of first byte of the encoded Multiple Record in terms of the write block size, as encoded in EBV-8 format in the multiple records directory.

10.1.53 Tag-Data-Profile-ID-Table

The application command argument **Tag-Data-Profile-ID-Table** identifies the table that shall be used as a reference for the sequence of **Object-Identifiers** and the compaction scheme to apply to the data **Object.**

10.1.54 Tag-Mask

The application command argument **Tag-Mask** represents a bit pattern in the **Inventory-EPC-UIImemory** command to enable tags with this particular bit pattern from the **Pointer** to be identified and returned. This field is incorporated directly into the associated ISO/IEC 18000-6C *Inventory* air interface command. It is also used in conjunction with **Length-Of-Mask**.

This argument is used with ISO/IEC 18000-6C when accessing EPCglobal related data.

10.1.55 TID-Segment-Map

The application response argument **TID-Segment-Map** is used as parameter to identify the raw byte string encoded as the TID, or **Singulation-Id**, on an ISO/IEC 18000-6 Type D tag.

10.1.56 UII-DSFID

The application argument **UII-DSFID** is used as parameter to identify the DSFID encoded with the UII segment of an ISO/IEC 18000-6 Type D tag.

10.1.57 UII-Segment-Map

The application response argument **UII-Segment-Map** is used as parameter to identify the raw byte string encoded within the UII segment of an ISO/IEC 18000-6 Type D tag.

10.1.58 Update-Multiple-Records-Directory

The application command argument **Update-Multiple-Records-Directory** is BOOLEAN, but the processing on the Logical Memory by the Data Processor needs to takes into account whether a directory already exists. The following states and processes apply:

- If a directory pre-exists and the command argument is set to TRUE, the directory is fully updated, including any previously missed directory entries.
- If a directory pre-exists and the command argument is set to FALSE, the argument is ignored and the directory is fully updated, including any previously missed directory entries.
- If no directory exists and the command argument is set to TRUE, the directory is created and fully updated, including any previously missed directory entries.
- If no directory exists and the command argument is set to FALSE, no directory is created.

10.1.59 Word-Count

The application command argument **Word-Count** identifies the number of words, counting from the **Word-Pointer,** that are to be read from an 18000-6C tag using the application command **Read-Words-Segmented-Memory-Tag**. This field is incorporated directly into associated ISO/IEC 18000-6C air interface commands.

10.1.60 Word-Pointer

The application command argument **Word-Pointer** identifies the first word to be read in a **Read-Words-Segmented-Memory-Tag** that returns an un-interpreted sequence of bytes on an ISO/IEC 18000-6C tag. This field is used in conjunction with the **Word-Count** field, and it is incorporated directly into an associated ISO/IEC 18000-6C air interface command.

10.2 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 **Completion-Code**, as specified in Table 7 — List of Completion-Codes. If errors occur, at least one **Completion-Code** shall be returned per response. Addressing multiple errors is beyond the scope of this International Standard, but multiple **Completion-Codes** may be provided depending on the design of the interface. Additionally, the "housekeeping" commands such as **Read-Logical-Memory-Map** and **Read-Words-Segmented-Memory-Tag** can be used for diagnostic purposes.

Table 7 — List of Completion-Codes

0	**No-Error**
1	**AFI-Not-Configured** For some reason, possibly because of a prior configure action, the command was not completed.
2	**AFI-Not-Configured-Locked** As (1) but responding that the existing **AFI** was also found to be locked.
3	**AFI-Configured-Lock-Failed** Successfully configured but not locked as required
4	**DSFID-Not-Configured** For some reason, possibly because of a prior configure action, the command was not completed.
5	**DSFID-Not-Configured-Locked** As (4) but responding that a pre-existing **DSFID** was found to be locked
6	**DSFID-Configured-Lock-Failed** Successfully configured but not locked as required.
7	**Object-Locked-Could-Not-Modify** The previous encodation on the RFID tag was locked, and as a result the attempt to update the **Object** value could not be completed.
8	**Singulation-Id-Not-Found** The particular RFID tag, specified by the **Singulation-Id**, could not be found in the operating area.
9	**Object-Not-Added** The **Object** and **Object-Identifier** plus associated syntax bytes could not be added to the Logical Memory Map (e.g. because there was insufficient memory space). This response also applies if the RFID tag supports a lock function that failed.
10	**Duplicate-Object** An **Object-Identifier** with the same value as the one to be processed (added, modified, or deleted) was found. The process was aborted.
11	**Object-Added-But-Not-Locked** The **Object** and **Object-Identifier** plus associated syntax bytes was added to the RFID tag that did not support a lock feature in the memory.
11	**Object-Not-Deleted** The **Object** and **Object-Identifier** plus associated syntax bytes could not be deleted from the Logical Memory Map (e.g. because the delete function is not supported by the particular RFID interrogator).
12	**Object-Not-Deleted** The **Object** and **Object-Identifier** plus associated syntax bytes could not be deleted from the Logical Memory Map (e.g. because the delete function is not supported by the particular RFID interrogator).

13	**Object-Identifier-Not-Found** The **Object-Identifier** intended to be processed was not actually encoded on the RFID tag.
14	**Object-Locked-Could-Not-Delete** The **Object** and **Object-Identifier** plus associated syntax bytes is locked and could not be deleted.
15	**Object-Not-Read** The intention to read the specified **Object** failed.
16	**Reserved**
17	**Blocks-Locked** This indicates that the intended action to erase encoded bytes from one or more blocks could not be actioned.
18	**Erase-Incomplete** 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.
19	**Read-Incomplete** The intention to read the complete contents of the Logical Memory Map failed, because not all blocks from the RFID tag were transferred to the Logical Memory.
20	**System-Info-Not-Read** The intention to read the system information failed.
21	**Object-Not-Modified** The **Object** and **Object-Identifier** plus associated syntax bytes could not be modified on the Logical Memory Map (e.g. because the modify function is not supported by the particular RFID interrogator).
22	**Object-Modified-But-Not-Locked** The **Object** was modified in the RFID tag that did not support a lock feature in the memory.
23	**Failed-To-Read-Minimum-Number-Of-Tags** The minimum number of RFID tags was not identified with the specified selection criterion, possibly because of a time-out.
24	**Failed-To-Read-Exact-Number-Of-Tags** The pre-defined exact number of RFID tags was not identified with the specified selection criterion, possibly because of a time-out.
25	**Password-Mismatch** The **Password** in the command failed to match the **Password** encoded on the RFID tag.
26	**Password-Not-Written** The **Password** in the command was not written to the RFID (e.g. this writing feature was not supported on the interrogator or RFID tag).
27	**Zero-Kill-Password-Error** The Kill operation was not executed because the Kill password has a zero value.
28	**Kill-Failed** The intended air interface *Kill* command failed (e.g. There was insufficient power to perform the kill operation).
29	**Object-Not-Editable** The **Object**, encoded as part of a **Packed-Object**, was not modified because it is not editable.
30	**Directory-Already-Defined** A **Packed-Object** already has a directory type defined.
31	**Packed-Object-ID-Table-Not-Recognised** The ID-Table called for in the command was not supported by the encoder or interrogator, and encoding is impossible to achieve.
32	**Tag-Data-Profile-ID-Table-Not-Recognised** The ID-Table called for in the command was not supported by the encoder or interrogator, and encoding is impossible to achieve.
33	**Insufficient-Tag-Memory** The operation failed because there was insufficient memory on the tag to satisfy the requested operation.
34	**AFI-Not-For-Monomorphic-UII** The operation failed because the **AFI** called for in the command is not registered for **Monomorphic-UIIs**.

35	**Monomorphic-UII-OID-Mismatch** The **Object-Identifier** defined in the command does not match that on the ISO/IEC 15961-2 Data Constructs register for this **AFI**.
36	**Command-Cannot-Process-Monomorphic-UII** A **Monomorphic-UII** was presented as an **Object** to command that does support this type of UII.
37	**Data-CRC-Not-Applied** At least one of the **Data-CRCs** requested in the **Extended-DSFID** was not applied to the data.
38	**Length-Not-Encoded-In-DSFID** At least one of the requested lengths was not encoded in the **Extended-DSFID**.
39	**Multiple-Records-Header-Not-Configured** Some of the processes required to configure the **Multiple-Records-Header** could not be processed by the Data Processor, resulting in an incomplete record.
40	**Multiple-Records-Header-Not-Locked** The **Multiple-Records-Header** was added to an RFID tag that did not support a selective lock feature in the memory.
41	**File-Support-Indicators-Not-Configured** Some of the processes required to configure the **File-Support-Indicators** could not be processed by the Data Processor, resulting in an incomplete encoding.
42	**File-Support-Indicators-Not-Locked** The **File-Support-Indicators** field was added to an RFID tag that did not support a selective lock feature in the memory.
43	**Data-Format-Not-Compatible-Multiple-Records-Header** The **Data-Format** in the command is at variance with the controlling **Data-Format** rules defined in the MR-header. The record was not encoded.
44	**Access-Method-Not-Compatible-Multiple-Records-Header** The **Access-Method** in the command is at variance with the controlling **Access-Method** rules defined in the MR-header. The record was not encoded.
45	**Sector-Identifier-Not-Compatible-Multiple-Records-Header** The sector identifier in the **Object-Identifier** in the command is at variance with the controlling sector identifier rules defined in the MR-header. The record was not encoded.
46	**Record-Preamble-Not-Configured** Some of the processes required to configure the record preamble could not be processed by the Data Processor, resulting in an incomplete encoding.
47	**Record-Preamble-Not-Locked** The record preamble was added to an RFID tag that did not support a selective lock feature in the memory.
48	**Multiple-Records-Directory-Not-Present** The command to read this directory could not be invoked because of the absence of the directory.
49	**Record-Not-Deleted-Preamble-Locked** The command could not be invoked because the record's preamble is locked.
50	**Record-Not-Deleted-Directory-Locked** The command could not be invoked because the record's entry in the directory is locked.
51	**Record-Not-Deleted-Lower-Level-Preamble-Locked**: The command could not be invoked because the preamble of a lower-level record is locked.
52	**Record-Not-Deleted-Encoding-Locked:** The command could not be invoked because part of the record is locked
53 to 252	**Unassigned**
253	**ISO/IEC 24791-5 Result-Code** This **Completion-Code** may be used to pass through the Result code from an ISO/IEC 24791-5 response in the form of {message code} {result value}.
254	**Undefined-Command-Error** An error occurred in a command, not defined by another code.
255	**Execution-Error** A system error occurred which made it impossible to action the command. The appropriate **Execution-Code** is returned in the response.

10.3 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 **Execution-Code**, as specified below. If systems errors occur, at least one **Execution-Code** shall be returned per response. Addressing multiple errors is beyond the scope of this International Standard, but multiple **Execution-Codes** may be provided depending on the design of the interface.

The following **Execution-Codes** apply, and are equivalent to those defined in ISO/IEC 15961-1.

0 **No-Error**: The command was executed without error

1 **No-Response-From-Tag**: No response was received from the RF tag

2 **Tag-Communication-Error**: The response(s) from the RFID tag(s) was corrupted (e.g. there was an aborted frame)

3 **Tag-CRC-Error**: A CRC error was detected in the RFID tag's response

4 **Command-Not-Supported**: The interrogator or RFID tag does not support the command.

5 **Invalid-Parameter**: The command parameter(s) are invalid

6 **Interrogator-Communication-Error**: An error occurred in the communication between the application and in the interrogator

7 **Internal-Error**: An error occurred in the application software

255 **Undefined-Error**: An error occurred, not defined by another code

11 Access-Method

The Access-Method, as defined by the application, is the most significant determinant of how data is encoded on the RFID tag. The value of Access-Method should be stored on the RFID tag, or may be defined by the air interface services, if this can be done unambiguously. The Access-Method is defined as an integer value in the application command and encoded as a compound bit value in the DSFID and the SFF (Special Features Flag) byte on the RFID tag.

Table 8 — Assigned and reserved Access-Methods defines the code structure.

Table 8 — Assigned and reserved Access-Methods

15961 integer code	15962 DSFID bit code	15962 SFF bit code	Name	Description
0	00	00	**No-Directory**	This structure supports the contiguous abutting of all the Data-Sets
1	01	00	**Directory**	The data is encoded exactly as for **No-Directory** but the RFID tag supports an additional directory, which is first read to point to the address of the relevant object identifier.
2	10	00	**Packed-Objects**	This is an integrated compaction and encoding scheme that formats data in an indexed structure as defined by the Application administrator (see ISO/IEC 15961-2)
3	11	00	**Tag-Data-Profile**	This is an integrated compaction and encoding scheme for a fixed set of data elements, each of a defined length

15961 integer code	15962 DSFID bit code	15962 SFF bit code	Name	Description
4	00	01	**Multiple-Records**	This encoding scheme enables multiple instances of **No-Directory**, **Packed-Objects**, and **Tag-Data-Profile** to be encoded on the same Logical Memory
5	00	10		reserved for future revisions of ISO/IEC 15962
6	00	11		reserved for future revisions of ISO/IEC 15962
7	01	01		reserved for future revisions of ISO/IEC 15962
8	01	10		reserved for future revisions of ISO/IEC 15962
9	01	11		reserved for future revisions of ISO/IEC 15962
10	10	01		reserved for future revisions of ISO/IEC 15962
11	10	10		reserved for future revisions of ISO/IEC 15962
12	10	11		reserved for future revisions of ISO/IEC 15962
13	11	01		reserved for future revisions of ISO/IEC 15962
14	11	10		reserved for future revisions of ISO/IEC 15962
15	11	11		reserved for future revisions of ISO/IEC 15962

NOTE In addition to the encoding schemes declared by the **Access-Method**, the **Monomorphic-UII** is an additional encoding scheme that is declared directly by the **AFI** (see 9.2.4).

The Extended Syntax Flag Byte 1 is required for all Access-Methods 4 to 15. It is only required for Access-Methods 0 to 3 to signal other functions supported by the RFID tag.

Each of the **Access-Methods** defines specific encoding rules for the **Object-Identifier** and for the data **Object**. The sub-clauses that follow describe each of the **Access-Methods** and define specific rules for encoding **Object-Identifiers** and data **Objects**.

Any of the **Access-Methods** may be applied to the user memory of a single memory structure RFID tag, subject to that **Access-Method** being supported by the particular application standard.

For Segmented memory structures, only some of the **Access-Methods** can be applied to particular memory banks. Details will be provided in the following sub-clauses.

11.1 No-Directory structure

The **Access-Method** = **No-Directory** is designed to achieve a combination of flexibility and efficiency for the bytes that are encoded on the RFID tag. In particular:

— Data **Objects** are compacted efficiently using a defined set of compaction techniques that reduce the encoding of data objects on the RFID tag across the air interface.

— Data formatting minimises the encoding of the **Object-Identifiers** on the RFID tag and on the air interface, but still provides complete flexibility for identifying specific data.

The encoding consists of a repeating cycle of **Data-Sets** (see 11.1.2), with each additional **Data-Set** abutted to the previous Data-Set as illustrated in Figure 4 — Logical Memory Schematic - No-Directory Structure.

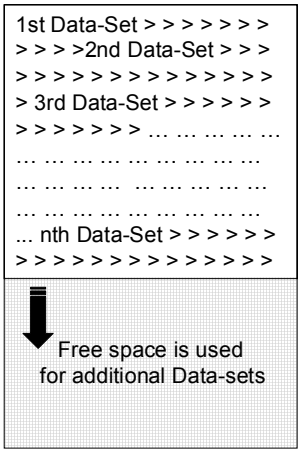


Figure 4 — Logical Memory Schematic - No-Directory Structure

The first **Data-Set** is encoded in the lowest block, subject to any requirements to encode the **AFI** and **DSFID** in the same encoding space.

11.1.1 Restrictions to air interfaces

This **Access-Method** may be used to encode ISO-related data in any memory area where applications are able to write a UII and/or item-related data, including the relevant memory banks in a segmented memory tag.

11.1.2 The Data-Set

Each **Data-Set** comprises the following component parts some of which are conditional on the value of the **Object-Identifier**:

- Precursor - This is typically 1 byte long and provides a bit-based meta-data structure of the encoded **Object-Identifier** and **Object**.
- Offset - This is only present when the **Data-Set** needs to be aligned on a block boundary, for example to ensure that the **Data-Set** is locked or not locked relative to adjacent **Data-Sets**.
- The length of the **Object-Identifier**.
- **Object-Identifier** - This is a formatted **Object-Identifier** that may be encoded as a **Relative-OID**, using the **Data-Format** to identify the **Root-OID** and reduce the number of encoded bytes, 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 Table 9 — Data-Set structures.

Table 9 — Data-Set structures

Description	Structure of Byte String for an Encoded Data Set						
Single Relative-OID 1 -14	Precursor	Length of data	Data ~~				
Single Relative-OID 15 -127	Precursor	Relative-OID	Length of data	Data ~~			
Other OID	Precursor	Length of OID	OID ~~	Length of data	Data ~~		
Single Relative-OID 1 –14 with Offset	Precursor	Offset	Length of data	Data ~~	Pad ~~		
Single Relative-OID 15 –127 with Offset	Precursor	Offset	Relative-OID	Length of data	Data ~~	Pad ~~	
Other OID with Offset	Precursor	Offset	Length of OID	OID ~~	Length of data	Data ~~	Pad ~~

NOTE ~~ indicates that this component typically can be multiple bytes,

The number of pad bytes could be 0 – 254, depending on the block size and lock block alignment

11.1.3 Encoding Rules

The encoding rules for this **Access-Method** are defined in Annex D, with the following annexes providing additional details:

Annex E provides detailed rules for the data compaction schemes.

Annex F provides details of the characters supported in each compaction scheme.

Annex G provides an encoding example.

11.2 Directory structure

The **Access-Method** = **Directory** has a two part structure in the Logical Memory:

— The lower addressed blocks are identical to the **No-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 5 — Logical Memory Schematic - Directory Structure illustrates this.

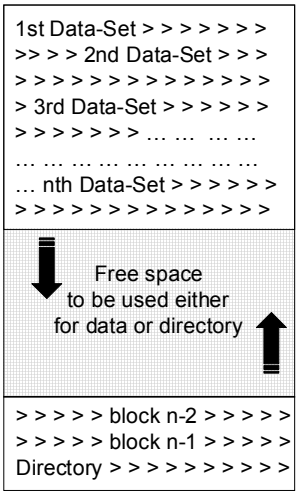


Figure 5 — Logical Memory Schematic - Directory Structure

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

It is possible to begin with a **No-Directory** Logical Memory structure, and some time later convert to a **Directory** structure. This first requires all of the content of the RFID tag's application memory to be transferred to the Logical Memory of the Data Processor. Then the **Access-Method** needs to be changed to indicate that a **Directory** structure is now invoked, and a directory created. Finally the **DSFID** needs to be updated on the RFID tag, and the content of the directory transferred to the RFID tag.

The directory should not be locked, because if so done, it renders it impossible to update **Data-Sets** in the **No-Directory** part of the Logical Memory. However, the **Data-Sets** in that region of memory may or may not be locked to suit the needs of the application.

11.2.1 Restrictions to air interfaces

For ISO/IEC 18000-6C RFID tags, this **Access-Method** shall only applied to Memory Bank 11_2. No other restrictions are known.

11.2.2 The Directory structure for Data-Format = {3 …287}

The Directory has the following structure of a repeating cycle for each **Data-Set** consisting of:

- The Precursor for the **Data-Set**.
- The length of the **Object-Identifier** (conditional).
- The **Object-Identifier** (conditional)
- The address as a byte value of the start of the precursor in the associated **Data-Set** within the **No-Directory** area.

The length of the **Object-Identifier** is conditional, because it is only required for **Relative-OIDs** greater than 127 or for full **Object-Identifiers**. The **Object-Identifier** is conditional because **Relative-OID** values in the range 1 to 14 are directly encoded in the Precursor.

The end of the Directory is signalled by the terminator byte, value 00_{16}, as defined in Annex D.3.

11.2.3 The Directory structure for Data-Format = 2

This Directory structure is identical to that as described in 11.2.2 except for the fact that the first element to be encoded refers to the **Root-OID** as follows:

- Precursor of the **Root-OID**.
- The length of the **Root-OID** (in bytes).
- The Root-OID.

There is no need to define the address of the **Root-OID**, because this is encoded in the first **Data-Set** with the first byte being at the lowest address in the **No-Directory** structure.

11.2.4 Encoding the address of the Data-Set

The address of the Precursor of each **Data-Set** in the **No-Directory** section of the Logical Memory is encoded in the directory using the length encoding rules defined in Annex 0. The address shall be based on the byte address of the memory, with the first byte in the first block being defined as byte value 00_{16}.

NOTE The byte address is used, because a **Data-Set** may begin or end at other than a block boundary.

11.2.5 Encoding Example

An example of the encoding with a directory is provided in Annex H.

11.3 Packed-Objects structure

The **Packed-Objects** encoding scheme has been developed since the first edition of this International Standard. It uses a rules-based table, defined by the application administrator and registered under the rules of ISO/IEC 15961-2. This encoding scheme specifies common compaction schemes which are significantly more efficient than those defined for the **No-Directory** structure, and a compaction scheme can be specified for each **Relative-OID** value in the table. In addition, the Packed-Objects encoding scheme may specify the use of the same compaction schemes as of the **No-Directory Access-Method**. This enables a simpler implementation, but still with encoding efficiencies over the basic **No-Directory Access-Method**. The basic process is illustrated in Figure 6 — The Packed-Objects encoding process.

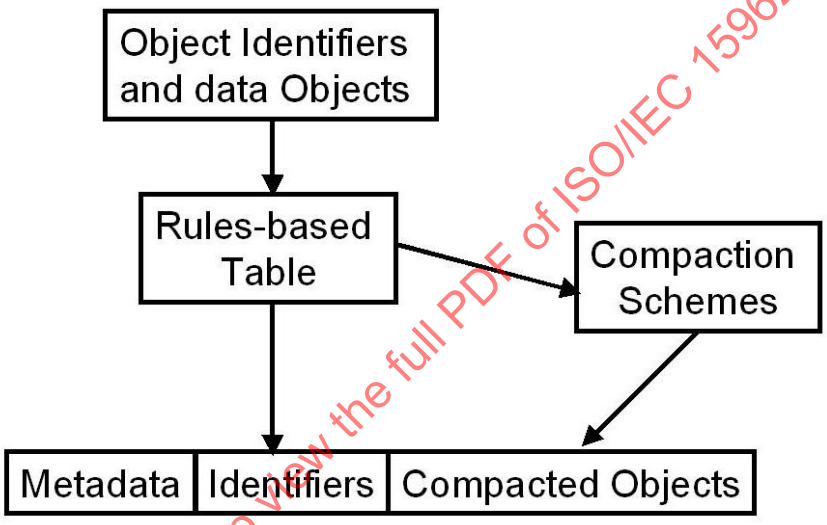


Figure 6 — The Packed-Objects encoding process

The **Object-Identifiers** and data **Objects** are input into the rules-based table, where the **Object-Identifiers** are separated from the **Objects**. The **Object-Identifiers** are encoded in a group, and the data is compacted in a group. The encoding sequence consists of metadata, an index of identifiers, followed by the compacted **Objects**. The index allows for all **Object-Identifiers** encoded within the **Packed-Object** to be identified, which enables a search for a particular **Object-Identifier** to be undertaken efficiently. If the required **Object-Identifier** is not present, then the search can continue in a subsequent **Packed-Object** structure, if any are encoded on the RFID tag.

The detailed of **Packed-Objects** are defined in the following annexes:

Annex I defines fully comprehensive Packed Object structure detailing all the mandatory and optional features

Annex J defines the ID tables

Annex K provides details of the encoding tables

Annex L provides the encoding rules

Annex M provides suggested decoding algorithms.

Not all features are mandatory, and a distinction will be made in relevant parts of the text of the annexes where optional components exist, and how the option is invoked.

This **Access-Method** may be used to encode ISO-related data in any memory area where applications are able to write a UII and/or item-related data, including the relevant memory banks in a segmented memory tag.

11.4 Tag Data Profile

The **Access-Method** = **Tag-Data-Profile** is designed to support applications that are able to define all the encoded data as mandatory and of a fixed or maximum length. It is possible to apply encoding rules that achieve an efficient encoding of the bytes on the RFID tag. In particular:

— The Tag-Data-Profile table, if accessed by the interrogator, provides a rapid access to any data on the RFID tag.

— Data **Objects** are compacted efficiently using a defined set of compaction techniques that reduce the encoding of data objects on the RFID tag across the air interface.

— The Precursor and other syntax on the **No-Directory Access-Method** are retained to enable interrogators with no access to the Tag-Data-Profile to decode the data.

The encoding consists of some meta-data that uniquely identifies the **Tag-Data-Profile** plus encoded data sets that are the same as if encoded using **the No-Directory Access-Method**, except that all the data has a predetermined length and compaction.

11.4.1 Restrictions to air interfaces

This **Access-Method** may be used to encode ISO-related data in any memory area where applications are able to write a UII and/or item-related data, including the relevant memory.

11.4.2 Defining the Tag-Data-Profile

A **Tag-Data-Profile** is identified by a rules-based table, defined by the application administrator and registered under the rules of ISO/IEC 15961-2. The table enables the encoder to convert the **Object-Identifiers**, as defined by the application, to a byte-mapped position on the Logical Memory. The table also specifies length, formats, compaction scheme and whether the Data **Object** is locked.

11.4.3 Encoding Rules

The ID Table and encoding rules for this **Access-Method** are defined in Annex N, with the following annexes providing additional details:

Annex O provides details of the structure of the Profile table.

Annex E provides detailed rules for the data compaction schemes.

Annex F provides details of the characters supported in each compaction scheme.

Annex P provides an encoding example.

11.5 Multiple-Records

The **Multiple-Records Access-Method** is designed to enable an application to encode more than one record in the same memory area of the RFID tag. For precision, the term Logical Memory is used in this clause and associated annexes to define the bounds of the memory that is being addressed. Logical Memory can therefore be applied to the entire encoding space of a tag with a single addressable memory, to a partitioned memory such as an addressable memory bank, or to a memory that is partitioned into files.

The first edition of this International Standard placed a restriction on having more than one instance of the same **Relative-OID** encoded in the same Logical Memory. The rules of the **Multiple-Records Access-**

Method overcome this restriction, but also enables distinct records, which use the encoding rules of other **Access-Methods**, to be distinguished one from another. The records may be all of the same type or of different types.

The basic structure is illustrated in Figure 7 — The basic structure for multiple records.

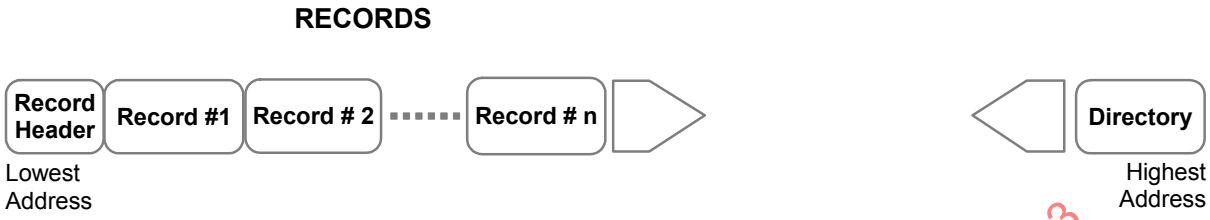


Figure 7 — The basic structure for multiple records

The **Access-Method** = **Multiple-Records** has a three part structure in the Logical Memory:

— A Multiple-Records header (hereafter MR-header) that is encoded from the lowest address in the memory. The MR-header needs to be of a sufficient size to support the number of initial and future records.

— One or more individual records, where each record fully complies with one of the following **Access-Methods: No-Directory, Packed-Objects,** and **Tag-Data-Profile**. The constraint is that the **Multiple-Records Access-Method** shall not contain the **Directory Access-Method** (1) or the **Multiple-Records Access-Method** (4) itself.

NOTE Future revisions to this International Standard might have the capability to support additional yet-to-be defined **Access-Methods** of the individual records.

— The higher blocks contain the directory of all of these records, as a series of directory entries, which may be added later. The directory entries are stored in reverse block sequence starting from the highest addressable (last) block, but each block is encoded from the lowest byte address to the highest byte address. This is similar to the ordering defined in 11.2 for the **Directory Access-Method**. Structuring the directory in this reverse sequence of blocks leaves unused blocks available either to encode additional records or directory until there are too few blocks remaining to encode new data.

As additional records are added, they are encoded in the next available memory block higher than the end of the previous record. The directory entries are made in the next available memory block lower than the previous end of the directory.

11.5.1 Categories of multiple records

11.5.1.1 Homogeneous Multiple-Records

For the purpose of this International Standard a homogeneous **Multiple-Records** encoding is one where all the records have the same **Data-Format**. This is illustrated schematically in Figure 8 — Example of a homogeneous Multiple-Records encoding, with the same domain.

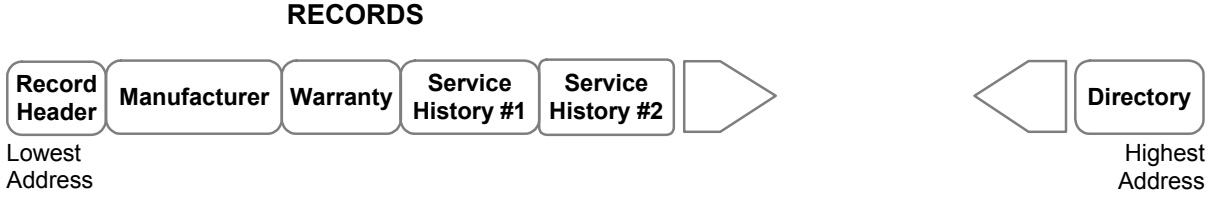


Figure 8 — Example of a homogeneous Multiple-Records encoding, with the same domain

Figure 8 — Example of a homogeneous Multiple-Records encoding, with the same domain illustrates a Logical Memory where data is associated with the same domain, but with the service history records being created by different individuals, even by different organisations, within the framework of that domain. This results in the same **Data-Format** for each record.

11.5.1.2 Heterogeneous Multiple-Records

For the purpose of this International Standard a heterogeneous **Multiple-Records** encoding is one where at least two records have a different **Data-Format**. This is illustrated schematically in Figure 9 — Example of a heterogeneous Multiple-Records encoding, with different domains.

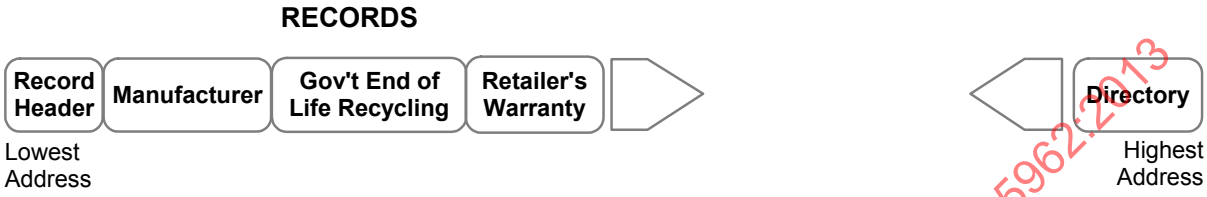


Figure 9 — Example of a heterogeneous Multiple-Records encoding, with different domains

Figure 9 — Example of a heterogeneous Multiple-Records encoding, with different domains illustrates a Logical Memory where data is encoded by organisations from three different domains. This results in different **Data-Formats** for some of the records.

11.5.1.3 Hierarchical records

This is a subset of the **Multiple-Records Access-Method** where hierarchical levels of data are logically dependent across the levels in the hierarchy, using a parent-child link between the records. The top-level records in the hierarchy may be either homogeneous or heterogeneous, but all lower-level records shall use homogeneous encoding.

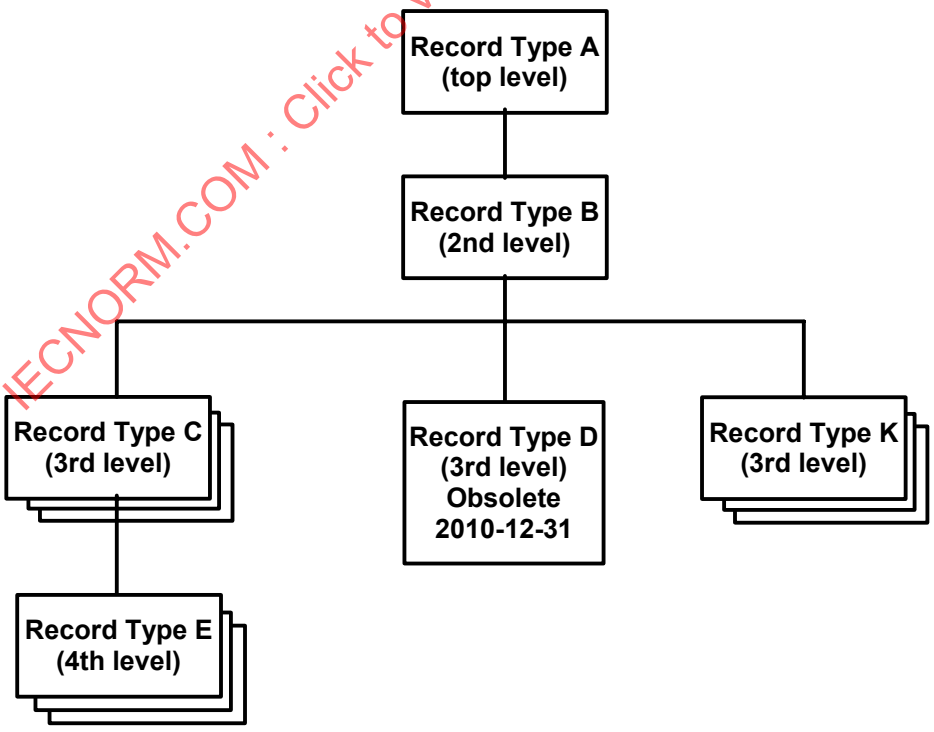


Figure 10 — Example of a hierarchy of records

The example in Figure 10 — Example of a hierarchy of records illustrates a set of hierarchical records schematically, and suggested style for the overview to be presented in an application standard. A hierarchical structure may have any number of levels greater than 1. There can only be one instance of the top-level record type. This example shows a second level record where there is only one instance, simply to illustrate that it is possible, for example an order record below a shipping record. The third layer in the example shows three different record types that are possible below the second level, and depending on the application some of these may be optional and not present in each hierarchical structure. The example shows that record types C and K can be included a number of times, but that record type D at this level can only occur once if included. The example shows that record type E is at the lowest (fourth) level and that each instance of record type C may in turn have only one instance of record type E (illustrated graphically by having no additional boxes).

The presence of record type K in the structure also illustrates that the application administrators may update the basic hierarchical structure by adding a completely new type of record at a later date. The reverse is also permitted (as illustrated with record type D), but such a record should be shown as "obsolete" on the hierarchical structure, because tags with the structure will still be in circulation after the final record of that type has been encoded.

This International Standard places the following constraints and freedoms in structuring a hierarchical record structure:

- Each record shall have a different hierarchical record number.
- The top-level record shall have the lowest number in the set.
- There are no constraints on the sequence for adding lower level records.
- There are no constraints on the number of record types in a hierarchy, or the number of records of each type.
- Each record type in the hierarchy may comprise of mandatory and optional data elements.
- No relative-OID may be encoded more than once in a given record.
- Records may be subsequently added to an existing hierarchy, but reference needs to be made to its parent record.

The structuring rules for multiple records enable many different hierarchical structures to be developed in application standards. Application administrators should define permitted structures to provide useful information to those encoding and decoding the records. Because the parent-child link is encoded, receiving systems can always reconstruct the intended without knowing the application standard, but knowledge of what records can establish such a relationship will ensure more efficient generation of commands and efficient responses.

11.5.1.4 Data element lists

A data element list contains a loop of a number of instances of the same single data element (e.g. the product codes of items in a container). Special encoding rules are used to enable what appears to be the encoding of multiple instances of the same **Object-Identifier** within the same record.

NOTE Having multiple instances of the same **Relative-OID** is not generally permitted in **Access-Methods** 0 to 3, but the mechanisms defined for a data element list enable this to be achieved.

EXAMPLE:

An airline container has 27 individual items of baggage. The data element has rules that enable each of these to be uniquely identified.

11.5.2 Object-Identifier structure

To support the processing of **Multiple-Records** on the interrogator, an implementation of the Data Processor, or at the application layer, the **Object-Identifier** shall follow one of three standardised structures. This also ensures that older implementations that cannot support **Multiple-Records** can carry out some basic processing.

11.5.2.1 The non-hierarchical Multiple-Record

The full **Object-Identifier** for a non-hierarchical record is:

1.0.15961.401. {data format = dictionary}.{sector identifier}.{record type}.{instance-of}.{Relative-OID of data element}

The arcs shown in bold text identify the **root-OID**. The following example describes the structure:

— The arc **401** identifies this as a **Multiple-Record** that is not part of a hierarchical structure.

— The arc for the **data format = dictionary** shall be the same as that for an existing registration for a **Data-Format.** For example, the arc =13 identifies the data dictionary for the mapping table for Data Identifiers.

NOTE This means that even if the registered data format has a root other than 1.0.15961 that the root defined in this clause applies for multiple records.

— The administrators of the data dictionary should assign a **sector identifier** to an administration that will define record types for a particular sector, e.g. a manufacturing sector using Data Identifiers. Further details are provided in 11.5.3.

— Thereafter, the sector administrators will be responsible for the assignment of a value for the arc for the record type for each defined record. The records shall be defined in a sector application standard, but do not need to be registered under ISO/IEC 15961-2 procedures.

— The instance-of arc is assigned by the user at the time that the record is created. Non-zero values indicate the encoding of the same record type (e.g. a maintenance record) at different points in time. A zero value instance-of arc in the OID structure is used to mean that there can be only one instance of this record in the logical memory.

— The Relative-OID of the data element arc is identical to that of the data element in the data dictionary, e.g. when using **Data-Format** 13, Relative-OID = 16 = Data Identifier 1P (Item Identification Code assigned by Supplier).

11.5.2.2 The hierarchical Multiple-Record

The full **Object-Identifier** for a hierarchical record is:

1.0.15961.402.{data format = dictionary}.{sector identifier}.{record type}.{hierarchical id}.{Relative-OID of data element}

NOTE See 11.5.2.3 for the data element list, which is a subset of the hierarchical record structure.

The arcs shown in bold text identify the root-OID. The following example describes the structure:

— The arc **402** identifies this as a **Multiple-Record** that is part of a hierarchical record structure.

— The arc for the **data format = dictionary** complies with the rules defined in 11.5.2.1 for data format = dictionary.

— The administrators of the data dictionary should assign a **sector identifier** to an administration that will define record types.

— Thereafter, the sector administrators will be responsible for assigning a value to the arc for the record type for each defined record. The records shall be defined in a sector application standard, but do not need to be registered under ISO/IEC 15961-2 procedures.

— In addition, The hierarchical parent – child relation as illustrated in Figure 10 — Example of a hierarchy of records shall be defined in the application standard.

— The hierarchical identifier arc is assigned by the user at the time that the record is created. The value 0 shall not be used to avoid ambiguity with a non-hierarchical record that has zero value instance-of arc.

— The value of the data element arc complies with the rules defined in 11.5.2.1 for for the Relative-OID of the data element.

11.5.2.3 The data element list

This is a hierarchical record that lists the same data element repeatedly. The full **Object-Identifier** for a data element list is:

1.0.15961.403.{data format = dictionary}.{sector identifier}.{record type}.{hierarchical id}.{Relative-OID of data element}.{list element number}

The arcs shown in bold text identify the root-OID. The following example describes the structure:

— The arc **403** identifies this as a **Multiple-Record** that is a data element list and part of a hierarchical record structure.

— The arc for the **data format = dictionary** complies with the rules defined in 11.5.2.1 for data format = dictionary.

— The administrators of the data dictionary should assign a **sector identifier** to an administration that will define record types.

— Thereafter, the sector administrators will be responsible for the assignment of a value for the arc for the record type for each defined record. The records shall be defined in a sector application standard, but do not need to be registered under ISO/IEC 15961-2 procedures.

— In addition, The hierarchical parent – child relation as illustrated in Figure 10 — Example of a hierarchy of records shall be defined in the application standard.

— The hierarchical identifier arc is assigned by the user at the time that the record is created. The value 0 shall not be used to avoid ambiguity with a non-hierarchical record that has zero value instance-of arc.

— The value of the data element arc complies with the rules defined in 11.5.2.1 for for the Relative-OID of the data element.

— The list element number is a sequentially assigned number to each data element encoded in the record.

11.5.3 The sector identifier

Three sector identifiers have specific meanings. The sector identifier = 0 is not used in the **Object- Identifier** structure because it is used in the encoding of the MR-header to indicate that the true sector identifieris defined in each record.

A sector identifier with the value = 1 shall be used to declare that the record types are for a closed system application. The data elements shall be compliant with the data dictionary.

A Sector identifier with the value = 2 shall be used in the following manner. Many data dictionaries have one or more primary keys identified by particular Relative-OID value(s). Typically such primary keys are encoded in the first position in a record, for example the Relative-OID = 19 in the Data Identifier data dictionary is for a CLEI code for telecommunications equipment. Other data elements are encoded that identify associated attributes of the particular CLEI coded item. To signal that the record contains a particular primary key data element, the record type following sector identifier= 2 shall have the same value as the first data element of the record, i.e., the same value as the record's primary key. So in the example, the following full **Object-Identifier** identifies that the CLEI code is the primary key:

1.0.15961.402. {data dictionary}.{sector identifier = 2}.{record type =19}.

Therefore, the administrators of the data dictionary shall assign sector identifiers incrementally starting with the value 3.

11.5.4 Restrictions to air interfaces

This **Access-Method** may be applied to any tag that has sufficient memory to support what is likely to be a larger requirement for encoding capacity. Specific additional constraints apply to tags with segmented memory. For the ISO/IEC 18000-6 Type C RFID tags the **Multiple-Records Access-Method** shall only be applied to Memory Bank 11_2. For the ISO/IEC 18000-6 Type D RFID tag, this **Access-Method** shall only applied be to the item-related data segment.

11.5.5 Encoding rules

The details of **Multiple-Records** defined in the following annexes:

Annex Q defines all the basic rules for homogeneous or heterogeneous **Multiple-Records** encoding.

Annex R defines the additional rules for hierarchical records.

An example of the encoding with a directory is provided in Annex S.

12 ISO/IEC 15434 direct encoding and transmission method using Access-Method 0 and Data-Format 3

The ISO/IEC 15434 direct encoding and transmission method is designed to support the encodation and transmission of a complete 15434 message in a compatible manner with similar messages in 2-D bar code symbols. This enables ISO/IEC 15434 messages to be used independently of the data carrier technology being selected.

Developers of application standards need to be aware this method requires the entire message to be read and parsed to extract a particular data element. There are alternative encoding schemes supported by the International Standard that support the encoding of individual data elements defined with an **Object-Identifier** that enable selective reading and writing between the application and the RFID tag.

12.1 General rules for ISO/IEC 15434 direct encoding

ISO/IEC 15434 specifies a syntax for high-capacity ADC media for a Message Envelope consisting of:

- a Message Header,
- one or more Format Envelope(s), and
- a Message Trailer (when required).

Each Format Envelope within the Message Envelope consists of:

— a Format Header,

— data, formatted according to the rules defined for that Format, and

— a Format Trailer (when required).

A “15434” message always starts with the following four character string for the message header:

[ ) > $^{R}_{S}$

The direct encoding method has rules to simplify the encoding, often enabling a specific 6-bit compaction scheme to be used. This codeset includes the 15434 control characters <RS>, <GS> and <EOT>. Also encoding 'shorthand' techniques are employed, for example to eliminate the need to encode the message header, yet still re-create this on the output transmission from the decoder.

Annex T defines the detailed encoding rules for ISO/IEC 15434 direct encoding and transmission method using Access-Method 0 and Data-Format 3.

12.2 Specific support for ISO TC122 standards

ISO TC 122 has a set of standards (currently ISO 17364, ISO 17365, ISO 17366, and ISO 17367) that only use ISO/IEC 15434 Format "06" messages encoding Data Identifiers. The character set and encoding rules for ISO TC 122 applications are a subset of the general rules for ISO/IEC 15434 direct encoding. As a result the same encoder and decoder processes are used. Compliant ISO 17364, ISO 17365, ISO 17366, and ISO 17367 applications need only to ensure that the correct subset of characters is used.

Annex U describes the ISO TC 122 differences from the general rules defined in Annex T.

13 Monomorphic-UII encoding

The **Monomorphic-UII** encoding scheme is designed to achieve a simple encoding of a Unique Item Identifier when encoded in a memory area dedicated to this function, and where no additional data is encoded in the same memory area. All of the features can be self-declaring through the registration of the particular **AFI** code values under the rules of ISO/IEC 15961-2. The following conditions apply:

— An **AFI** assigned to a particular **Monomorphic-UII** shall not support the encoding of any additional item-related data. If item-related data is required for the application, then this shall use an entirely different **AFI** and **Data-Format**.

— The **Monomorphic-UII** is therefore only capable of being encoded in a tag memory architecture that supports the separate encoding of a UII or as a single data object in a monolithic structured memory tag.

— The **AFI** fully defines all the arcs of the **Object-Identifier** for communications in the commands of ISO/IEC 15961 and in the ISO/IEC 24791 standards.

— The **Monomorphic-UII** does not require the encoding of a **DSFID**

Each **AFI** shall declare which of the encoding schemes, as defined in the following sub-clauses, is used. In addition, the registration of the AFI in compliance with ISO/IEC 15961-2 shall define all the necessary additional parameters, including: the **Object-identifier** for the UII, whether the UII is fixed length or variable length, and the length in bytes. If the tag architecture does not support a method of declaring the length of the encoded UII then the fact that this is required as a preamble to the encoded UII shall be part of the registration.

13.1 6-bit encoding

If the **AFI** declares that 6-bit encoding is used for the particular UII, the character set shall be limited to characters 20_{16} to $5F_{16}$ as defined in Annex F. The UII shall be encoded using the 6-bit encoding scheme as defined in Annex E.4.

If the tag architecture supports a method to declare the size of encoding of the UII memory, there is no requirement to encode any length indicator on the RFID tag, even if the UII may be of different lengths. If the tag architecture does not support a method to declare the size of encoding of the UII memory, then the compacted UII shall be preceded by a single byte indicating the length, in bytes, of the compacted UII.

The end of the encoding is defined by the 6-bit encoding rules, with one minor variation. If the tag architecture supports an encoding unit of more than an 8-bit byte (for example ISO/IEC 18000-6C uses a 16-bit word) then of the pad string “100000” is repeated as necessary in full and then truncated to fill the encoding space. Using the example of encoding on a 16-bit word structure, here are the eight end conditions:

- Exactly on the boundary – no action
- Requiring two pad bits – encode 10
- Requiring four pad bits – encode 1000
- Requiring six pad bits – encode 100000
- Requiring eight pad bits – encode 10000010
- Requiring ten pad bits – encode 1000001000
- Requiring twelve pad bits – encode 100000100000
- Requiring fourteen pad bits – encode 10000010000010

The decoding rules as defined in Annex E.4 apply so that if pad strings “10”, “1000” or “100000” are present at the end of the encoded bit string, they are discarded. For a Monomorphic-UII there can be repeated patterns, as shown above, that need to be discarded.

13.2 7-bit encoding

If the **AFI** declares that 7-bit encoding is used for the particular UII, the character set shall be limited to characters 20_{16} to $7E_{16}$ as defined in Annex F. The UII shall be encoded using the 7-bit encoding scheme as defined in Annex E.5.

If the tag architecture supports a method to declare the size of encoding of the UII memory, there is no requirement to encode any length indicator on the RFID tag, even if the UII may be of different lengths. If the tag architecture does not support a method to declare the size of encoding of the UII memory, then the compacted UII shall be preceded by a single byte indicating the length, in bytes, of the compacted UII.

The end of the ending is defined by the 7-bit encoding rules, with as many 1 bits as necessary to complete the encoding space. For example, if a 16-bit encoding structure is being used and only the first bit of the final word is part of the encoding, this shall be followed by the bit string "111111111111111". During the decode process this will be identified as two complete pad characters followed by a single 1 bit. All of which are discarded.

13.3 URN Code 40 encoding

This particular encoding is designed to support the encoding of a hierarchical URN with the appropriate separators being the hierarchical components. Therefore, when the URN is decoded from the RFID tag it is presented in a structure that is compatible with that required for resolving on the Internet.

If the **AFI** declares that URN Code 40 encoding is used for the particular UII, the character set shall be as defined in Annex V, including any required UTF-8 character codes. The UII shall be encoded using the URN Code 40 encoding defined in Annex V.

13.4 8859-1 octet encoding

If the **AFI** declares that 8-bit encoding is used for the particular UII, the character set shall be any ISO/IEC 8859-1 character, including control characters, in the range 00_{16} to FF_{16}.

If the tag architecture supports a method to declare the size of encoding of the UII memory, there is no requirement to encode any length indicator on the RFID tag, even if the UII may be of different lengths. Any additional bytes required to meet the air interface length (e.g. the length of a word) shall comprise of the necessary number of bytes 00_{16}, the first of which is processed as a terminator.

If the tag architecture does not support a method to declare the size of encoding of the UII memory, then the compacted UII shall be preceded by a single byte indicating the length, in bytes, of the compacted UII.

13.5 Application-defined 8-bit coding

If 8-bit encoding is used, including any external compaction, that is different from an ISO/IEC 8859-1 interpretation it shall be declared as application-defined. In this case, the character set, encoding scheme and decoded interpretation are defined by the application and beyond the scope on this International Standard.

NOTE The output from the decoder transfers the byte string and declares this as Application-Defined.

Annex A
(informative)

Air interface support for application commands

A.1 Overview

As indicated in 8.5, different air interface protocols have different degrees of compatibility with individual application commands. Details are set out below for some of the air interface protocols. If an air interface protocol is not covered, then some of the remainder of this annex, in combination with the air interface protocol standard, could still be helpful to implementers developing solutions.

A.2 ISO/IEC 18000-3 Mode 1 support

ISO/IEC 18000-3 Mode 1 Rev 1 provides mandatory and optional commands (some of which are defined as requirements for RFID for Item Management) that support the application command as detailed in Table A.1 — ISO/IEC 18000-3 Mode 1 support for application commands.

Table A.1 — ISO/IEC 18000-3 Mode 1 support for application commands

Code	Application Command Name	Supported	Qualified Support
1	Configure-AFI	YES	
2	Configure-DSFID	YES	
3	Inventory-Tags	YES	
5	Delete-Object	YES	
6	Modify-Object	YES	
8	Read-Object-Identifiers	YES	
10	Read-Logical-Memory-Map	YES	
12	Erase-Memory	YES	
13	Get-App-based-System-Info	YES	
17	Write-Objects	YES	
18	Read-Objects	YES	
19	Write-Objects-Segmented-Memory-Tag	NO	
20	Write EPC-UII	NO	
21	Inventory-ISO-UIImemory	NO	
22	Inventory-EPC-UIImemory	NO	
23	Write-Password-Segmented-Memory-Tag	NO	
24	Read-Words-Segmented-Memory-Tag	NO	
25	Kill-Segmented-Memory-Tag	NO	
26	Delete-Packed-Object	YES	
27	Modify-Packed-Object-Structure	YES	
28	Write-Segments-6TypeD-Tag	NO	
29	Read-Segments-6TypeD-Tag	NO	
30	Write-Monomorphic-UII	YES	
31	Configure-Extended-DSFID	YES	
32	Configure-Multiple-Records-Header	YES	
33	Read-Multiple-Records	YES	
34	Delete-Multiple-Record	YES	

Table A.1 — ISO/IEC 18000-3 Mode 1 support for application commands indicates that application commands are either supported, or not supported. Therefore, there is no requirement to create an additional interface mechanism to support the application commands that have "NO" in the supported column.

A.3 ISO/IEC 18000-6 Type C support

The support by ISO/IEC 18000-6C for the application commands is based on ISO/IEC 18000-6 AMD1:2006. Table A.2 — ISO/IEC 18000-6 Type C support for application commands lists whether an application command can be supported or not, and identifies those that can be supported in a qualified manner. Details of this qualified support, or alternative solutions, are provided in the following sub-clauses.

Table A.2 — ISO/IEC 18000-6 Type C support for application commands

Code	Command Name	Supported	Qualified Support
1	Configure-AFI	NO	YES
2	Configure-DSFID	NO	YES
3	Inventory-Tags	NO	YES
5	Delete-Object	YES	
6	Modify-Object	YES	
8	Read-Object-Identifiers	YES	
10	Read-Logical-Memory-Map	NO	YES
12	Erase-Memory	YES	
13	Get-App-Based-System-Info	NO	YES
17	Write-Objects	NO	YES
18	Read-Objects	YES	
19	Write-Objects-Segmented-Memory-Tag	YES	
20	Write EPC-UII	YES	
21	Inventory-ISO-UIImemory	YES	
22	Inventory-EPC-UIImemory	YES	
23	Write-Password-Segmented-Memory-Tag	YES	
24	Read-Words-Segmented-Memory-Tag	YES	
25	Kill-Segmented-Memory-Tag	YES	
26	Delete-Packed-Object	YES	
27	Modify-Packed-Object-Structure	YES	
28	Write-Segments-6TypeD-Tag	NO	
29	Read-Segments-6TypeD-Tag	NO	
30	Write-Monomorphic-UII	YES	
31	Configure-Extended-DSFID	YES	
32	Configure-Multiple-Records-Header	YES	
33	Read-Multiple-Records	YES	
34	Delete-Multiple-Record	YES	

A.3.1 Configure-AFI

The preferred means of encoding the **AFI** for this air interface is to use the **Write-Objects-Segmented-Memory-Tag** command, which incorporates the **AFI** as part of the sequence of bytes to be encoded in the RFID tag. This is because the **AFI** is a single byte, and the minimum communication across the ISO/IEC18000-6C air interface is a 16-bit word. Invoking the **Configure-AFI** command on its own will require the 16-bit word in memory to be overwritten at some subsequent stage.

A.3.2 Configure-DSFID

The preferred means of encoding the **DSFID** for this air interface is to use the **Write-Objects-Segmented-Memory-Tag** command, which incorporates the **DSFID** as part of the sequence of bytes to be encoded in the

RFID tag. This is because the **DSFID** is generally a single byte, and the minimum communication across the ISO/IEC18000-6C air interface is a 16-bit word. Invoking the **Configure-DSFID** command on its own will require the 16-bit word in memory to be overwritten at some subsequent stage.

A.3.3 Inventory-Tags

The **Inventory-Tags** command is intended to return a unique **Singulation-Id** to enable ongoing communications with the RFID tag. For many air interface protocols, this is a permanently encoded code in a dedicated part of memory on the RFID tag. ISO/IEC 18000-6C uses different methods to maintain communication, a dynamically created value.

If there is a requirement to read the optional unique **Singulation-Id** encoded on an 18000-6C tag, then the **Read-Words-Segmented-Memory-Tag** command may be used instead, with the **Memory-Bank** argument set = 10.

If the requirement is to read a unique item identifier, then the **Inventory-ISO-UIImemory** or **Inventory-EPC-UIImemory** command may be used, depending on the domain of the application. These commands return the content of the UII memory, which is required to be unique.

A.3.4 Read-Object-Identifiers

The **Read-Object-Identifiers** command returns a set of **Object-Identifiers** encoded on the RFID tag. Multiple memory banks were not supported when this command was first defined (in ISO/IEC 15962:2004).

Because ISO/IEC 18000-6C does not support an air interface command to read the content of different memory banks simultaneously, one way to invoke this application command is to apply it only using the **Memory-Bank** argument = 11. If there is an additional requirement to read any **Object-Identifier(S)** from Memory Bank 01, then the **Inventory-ISO-UIImemory** command should be used.

A.3.5 Read-Logical-Memory-Map

The **Read-Logical-Memory-Map** command was designed to read the encoded bytes from a single flat file memory on an RFID tag, and not a segmented memory as used with ISO/IEC 18000-6C tags. The **Read-Words-Segmented-Memory-Tag** command should be used in preference, setting the selected **Memory-Bank** argument for which information is required.

A.3.6 Get-App-Based-System-Info

The **Get-App-Based-System-Info** command is primarily intended to support tags where there is a directly equivalent air interface command. The information that is returned is the **AFI** and **DSFID**. An alternative way to obtain this information (possibly with other data) is to use the **Inventory-ISO-UIImemory** command to return the **AFI** and **DSFID** from memory bank = 01. The **Read-Words-Segmented-Memory-Tag** command can be used to read the **DSFID** for memory bank = 11.

NOTE No AFI code value is encoded in memory bank = 11.

A.3.7 Write-Objects

The **Write-Objects** command may be implemented with ISO/IEC 18000-6C, if it is restricted to memory bank = 11. The **Write-Objects-Segmented-Memory-Tag** command should be used in preference, because this contains all the relevant arguments needed to interface with an ISO/IEC 18000-6C RFID tag.

If the Object-Lock argument (see 10.1.38) is invoked for any data element, the implementation depends on the capability of the individual RFID tag, ranging for selective locking of part of memory bank 11 to only being able to lock a complete memory bank. The capability to lock data on an ISO/IEC 18000-6 Type C tag is defined in Annex C.8.5.

The following rules apply to locking memory:

a) If the tag is complaint with ISO/IEC 18000-6C AMD1, locking is only allowed for the complete memory bank. Therefore, if the **Write-Objects-Segmented-Memory-Tag** command contains one or more **object-lock** arguments that are contradictory (i.e. some TRUE and some FALSE arguments), or if the command specifies a lock status that differs from the lock status of encoding on the memory bank, then the entire command shall be considered in error.

b) If the tag is compliant with a version of ISO/IEC 18000-6C later than AMD1 then selective locking is an optional feature, which is declared by each tag model. There are sub-rules for encoding as follows in MB11:

    i) If the feature is supported, and the encoding is for MB11, then selective locking shall be applied as determined by the **object-lock** arguments for each data element.

    ii) If the feature is not supported on the RFID tag for MB11, but locking is called for when only a single data element is to be encoded, or for all data elements in a consistent manner, then the entire memory bank shall be locked.

    iii) If the feature is not supported on the RFID tag for MB11, and the **object-lock** arguments are inconsistent, then the entire command shall be considered in error.

c) The selective locking feature is not supported for MB01,so the following rules apply:

    i) If a single data element is to be encoded in MB01, then the **object-lock** argument shall be applied to all of MB01

    ii) If multiple data elements are to be encoded and all have the same **object-lock** argument, then this shall be applied to all of MB01.

    iii) If multiple data elements are to be encoded and these have different **object-lock** arguments the entire command shall be considered in error.

A.4 ISO/IEC 18000-6 Type D support

The support by ISO/IEC 18000-6 Type D for the application commands is based on ISO/IEC 18000-6. This air interface differs from others in ISO/IEC 18000-6 in that it uses a modified tag talks first protocol. To avoid interference with other the air interface protocols no forward link is specified in that standard. This means that any encoding of data has to be achieved using custom air interface commands, and in turn there are no application commands specified in this International standard for writing data to the tag. Table A.3 — ISO/IEC 18000-6 Type D support for application commands lists whether an application command can be supported or not, and identifies those that can be supported in a qualified manner. Details of this qualified support, or alternative solutions, are provided in the following sub-clauses.

Table A.3 — ISO/IEC 18000-6 Type D support for application commands

Code	Command Name	Supported	Qualified Support
1	Configure-AFI	NO	
2	Configure-DSFID	NO	
3	Inventory-Tags	YES	
5	Delete-Object	NO[see note]	
6	Modify-Object	NO[see note]	
8	Read-Object-Identifiers	YES	
10	Read-Logical-Memory-Map	YES	
12	Erase-Memory	NO[see note]	
13	Get-App-Based-System-Info	NO	YES

Code	Command Name	Supported	Qualified Support
17	Write-Objects	NO[see note]	
18	Read-Objects	YES	
19	Write-Objects-Segmented-Memory-Tag	NO[see note]	
20	Write EPC-UII	NO	
21	Inventory-ISO-UIImemory	YES	
22	Inventory-EPC-UIImemory	NO	
23	Write-Password-Segmented-Memory-Tag	NO	
24	Read-Words-Segmented-Memory-Tag	NO	
25	Kill-Segmented-Memory-Tag	NO	
26	Delete-Packed-Object	NO[see note]	
27	Modify-Packed-Object-Structure	NO[see note]	
28	Write-Segments-6TypeD-Tag	NO[see note]	
29	Read-Segments-6TypeD-Tag	YES	
30	Write-Monomorphic-UII	NO[see note]	
31	Configure-Extended-DSFID	NO[see note]	
32	Configure-Multiple-Records-Header	NO[see note]	
33	Read-Multiple-Records	YES	
34	Delete-Multiple-Record	NO[see note]	

NOTE Commands 5, 6, 12, 17, 19, 26, 27, 28, 30, 31, 32 and 33 need to be supported by custom air interface commands provided by the vendors of ISO/IEC 18000-6 Type D tags. These particular application commands may be used as a model for defining data to be encoded on a Type D tag. This is a realistic approach if the Type D air interface commands for the forward linked are modelled on the Type C forward link commands.

A.4.1 Configure-AFI

A customs method of encoding the **AFI** for this air interface is to base it on the **Write-Objects-Segmented-Memory-Tag** command, which incorporates the **AFI** as part of the sequence of bytes to be encoded in the RFID tag. This is because the **AFI** is a single byte, and the minimum communication across the ISO/IEC18000-6 Type D air interface is a 16-bit word. Invoking the **Configure-AFI** command on its own will require the 16-bit word in memory to be overwritten at some subsequent stage.

A.4.2 Configure-DSFID

A customs method of encoding the **DSFID** for this air interface is to base it on the **Write-Objects-Segmented-Memory-Tag** command, which incorporates the **DSFID** as part of the sequence of bytes to be encoded in the RFID tag. This is because the **DSFID** is generally a single byte, and the minimum communication across the ISO/IEC18000-6 Type D air interface is a 16-bit word. Invoking the **Configure-DSFID** command on its own will require the 16-bit word in memory to be overwritten at some subsequent stage.

A.4.3 Inventory-Tags

The **Inventory-Tags** command is intended to return a unique **Singulation-Id** (known as a TID-S and defined in Annex C.9.1) to enable ongoing communications with the RFID tag or to determine a population of tags within the reading zone. The ISO/IEC 18000-6 Type D unique **Singulation-Id** is always encoded from the first page of memory.

Because the **Singulation-Id** structure is pre-defined and based on the value of the lead byte, it is possible to carry out an inventory of Type D tags with any mixture of **Singulation-Id** structures.

A.4.4 Get-App-Based-System-Info

The **Get-App-Based-System-Info** command is primarily intended to support tags where there is a directly equivalent air interface command. The information that is returned is the **AFI** and **DSFID**. An alternative way to obtain this information (possibly with other data) is to use the **Inventory-ISO-UIImemory** command to

return the **AFI** and **DSFID** from the UII segment. The **Read-Segments-6TypeD-Tag** command can be used to read the **DSFID** for from the item-related segment and the AFI and DSFID from the UII segment.

NOTE No AFI code value is encoded in the Item-related data segment.

A.4.5 Read-Segments-6TypeD-Tag

The **Read-Segments-6TypeD-Tag** command is used to subdivide the encoding from an ISO/IEC 18000-6 Type D into its segments and further subdivisions. This is possible because the air interface protocol delivers the complete encoding from the tag in a single transaction.

As this command is currently only specific to this tag details are specified in 8.3.7.

Annex B
(normative)

Pro forma description for the Tag Driver

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

B.1 Defining the Singulation-Id

The **Singulation-Id** is the device in the Data Protocol to link the RFID tag to the Logical Memory. Clause 9.2.1 identifies some of the possible methods that can be invoked on the RFID tag, the air interface, or even using unique encoded data to achieve this.

The Tag Driver shall define the technique that the RFID tag uses to create the **Singulation-Id**.

B.2 System information : AFI

The **AFI** (see 9.2.4) is a prime selection mechanism for specifying a relevant subset of RFID 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 RFID tags supports the **AFI** 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.

B.3 System information: DSFID

The **DSFID** (see 9.2.5) incorporates the **Access-Method** to define the structure of the bytes on the RFID tag's Logical Memory Map, and incorporates the **Data-Format** to define particular sets of encoded application data. It is a key to the Data Processor on how to organise selective transactions defined by the application commands and how to condense the **Object-Identifiers** during encoding and to expand it during decoding. The **DSFID** is one byte long and its value shall be defined by the application.

The Tag Driver shall define how the RFID tags support the **DSFID** 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 **Access-Methods** 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.

An extension mechanism for the **Data-Format** is defined in 9.2.6 and a separate extension mechanism for the **Access-Method** and other tag functions is defined in 9.2.7. If these extension mechanisms are supported, they shall be encoded as defined for the individual Tag Drivers in Annex C.

B.4 Memory-related parameters

B.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 Processor.

B.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.

B.4.3 Number of blocks

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

B.4.4 Memory mapping

The Tag Driver shall specify:

1. Whether the memory is structured as a single 'flat' file or is organised in multiple, separately addressable, partitioned zones (or memory banks).

2. The location within the memory (as defined in ISO/IEC 18000) where the Logical Memory Map begins. This requires additional rules for addressing partitioned zones.

3. 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.

4. 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.

5. 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 Processor.

B.5 Support for commands

The manner in which the application commands are presented in ISO/IEC 15961-1 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 **Object-Lock** might not be supported by the RFID tag command calls; therefore the appropriate **Completion-Codes** shall be used.

c. be unable to support the functionality of a complete application command, for example to modify or delete an **Object-Identifier** and **Object** on an RFID 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 RFID tag command) are linked. Precise coding rules are not required and are to be resolved at the implementation level.

NOTE Annex A provides an overview of the relevance of an application command to particular air interface protocols, but does not address the detail of air interface commands that are optional either in the RFID tag or the interrogator.

Annex C
(normative)

ISO/IEC 18000 Tag Driver Descriptions

C.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.

C.1.1 Defining the Singulation-Id

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.

C.1.2 System information: AFI

The **AFI** represents the type of air interface application targeted by the Interrogator and is used to extract only those tags meeting the required selection criterion from all the tags present. The **AFI** may be programmed and locked by the respective commands. The **AFI** is coded on one byte.

The support of **AFI** by the tag is optional. If the **AFI** is not supported by the RFID 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_{16} irrespective of the AFI value on the tag. The tag shall also respond if the **AFI** flag is not set in the command.

C.1.3 System information : DSFID

The **DSFID** 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'.

C.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.

C.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.

C.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.

C.2.1 Defining the Singulation-Id

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

C.2.2 System information: AFI

The **AFI** is supported in ISO/IEC 18000-3 Mode 1 through the **AFI** mechanism, which is a separate register in the tag memory. 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 Annex C.2.5), the RFID 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.

C.2.3 System information : DSFID

The **DSFID** is an 8 bit code and is supported in ISO/IEC 18000-3 Mode 1 through one of these mechanisms:

A. the **DSFID** air interface mechanism, which is a separate register in the tag memory (see Annex C.2.3.1); or

B. the **DSFID** encoding rule, which requires the **DSFID** to be encoded and read using general air interface write and read commands (see C.2.3.2).

C. Option A is preferred; but option B is provided for ISO/IEC 18000-3 Mode1 tags that cannot support option A. Details of the Singulation-Ids (i.e. the first three bytes of the 64-bit code defined in C.2.1) that support one or other option are available on a JTC1 SC31 WG4 Standing Document.

C.2.3.1 The DSFID air interface mechanism

The **DSFID** is supported in some ISO/IEC 18000-3 Mode 1 RFID tags through the **DSFID** air interface mechanism, which is a separate register in the tag memory supported by specific commands. **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 (see Annex C.2.3.2), the tag shall return **DSFID** = '00' in response to the air interface *Inventory* command.

C.2.3.2 The DSFID encoding rule

If the **DSFID** air interface mechanism is not available, then the **DSFID** shall be encoded as part of the encoded byte stream in the first byte of the user memory. Used in this way, the **DSFID** cannot be returned

during the air interface inventory process, and cannot be programmed and locked by specific air interface commands. The only way the **DSFID** can be locked is by locking the first block, but this has implications for locking of encoded data.

The manufacturer and models of tags supported by this mechanism are defined in the Standing Document, mentioned above.

C.2.3.3 Extensions mechanisms for the Data-Format, Access-Method, and other tag functions

If the RFID tag supports the DSFID air interface mechanism, then all of the required extension bytes defined in 9.2.6 to 9.2.17 shall be encoded starting from the first byte of user memory and precede any encoded packets of data.

If the RFID tag supports the DSFID encoding rule, then all of the required extension bytes defined in 9.2.6 to 9.2.17 shall be encoded starting from the second byte of user memory and precede any encoded packets of data.

C.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 available on a JTC1 SC31 WG4 Standing Document, or can be returned by the air interface *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.

C.2.5 Support for commands

Only two air interface 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.

ISO/IEC 18000-3 Rev 1 now includes a new class of air interface command status, indicating that a command is "required for item management". The details of this status are included in Annex G of that standard. Table C.1 — Required commands and their codes provides a summary of the required commands.

Table C.1 — Required commands and their codes

Command Code	ISO/IEC 18000-3 Mode 1 BasicType	Function	Item Management Requirement
'01'	Mandatory	Inventory	The AFI is a requirement in the command, and the DSFID is required as part of the response
'02'	Mandatory	Stay quiet	No change
'20'	Optional	Read single block	The interrogator shall support this command. The RFID tag shall support this command if the *Read multiple blocks* command is not supported

Command Code	ISO/IEC 18000-3 Mode 1 BasicType	Function	Item Management Requirement
'21'	Optional	Write single block	The interrogator shall support this command. The RFID tag shall support this command if the *Write multiple blocks* command is not supported
'22'	Optional	Lock block	Required for the interrogator and for the RFID tag.
'23'	Optional	Read multiple blocks	The interrogator shall support this command. The RFID tag shall support this command if the *Read single block* command is not supported
'24'	Optional	Write multiple blocks	The interrogator shall support this command. The RFID tag shall support this command if the *Write single block* command is not supported
'25'	Optional	Select	this command shall be supported in interrogators and should be supported in tags
'26'	Optional	Reset to ready	this command shall be supported in interrogators and should be supported in tags
'27'	Optional	Write Afi	Required for the interrogator and for the RFID tag.
'28'	Optional	Lock Afi	Required for the interrogator and for the RFID tag.
'29'	Optional	Write DSFID	Required for the interrogator and for the RFID tag.
'2A'	Optional	Lock DSFID	Required for the interrogator and for the RFID tag.
'2B'	Optional	Get system information	Required for the interrogator and for the RFID tag.
'2C'	Optional	Get multiple block security status	Required for the interrogator and for the RFID tag.

C.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 **Singulation-Id** of the tag to be selected. All further commands shall have their Select flag set, but do not contain the **Singulation-Id**, therefore saving transmission time. Individual commands with other **Singulation-Ids** can be sent to specific tags during the sequence. See ISO/IEC 18000-3 Mode 1 for details.

C.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.

C.3.1 Defining the Singulation-Id

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.

C.3.2 System information: AFI

The **AFI** 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 $FFFF_{16}$.

C.3.3 System information : DSFID

The protocol describes Hardcode fields that provide a function similar to system information, and include the **DSFID**. 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.

C.3.4 Memory-related parameters

The RFID 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.

C.3.5 Support for commands

The RFID 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).

C.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.

C.4.1 Defining the Singulation-Id

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.

C.4.2 System information: AFI

The **AFI** 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 $0A_{16}$ 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.

C.4.3 System information : DSFID

The **DSFID** is supported in ISO/IEC 18000-4 Mode 1 through the **DSFID** mechanism. The **DSFID** indicates how the application data is structured in the tag application memory. As defined in Annex C.4.2, bytes 12 through 17 are reserved in tag system memory for representation of tag memory system information. Byte 12 with the value $0A_{16}$ 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'.

C.4.4 Memory-related parameters

The first 18 bytes (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 is 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.

C.4.5 Support for commands

Not all ISO/IEC 18000-4 Mode 1 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.

C.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.

C.5.1 Defining the Singulation-Id

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

C.5.2 System information: AFI

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

C.5.3 System information: DSFID

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

- Access-Method = No-Directory
- Data-Format = Full-Feature

C.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

C.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 cannot be executed.

C.6 Tag Driver for ISO/IEC 18000-6 Type A: 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 RFID tags compliant to ISO/IEC 18000-6 Type A.

C.6.1 Defining the Singulation-Id

Each tag is identified uniquely by a UID of 64 bits or an SUID of 40 bits. 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 for Type A is specified in ISO/IEC 18000-6 in Clause 8.1.2

C.6.2 System information: AFI

The **AFI** is supported in ISO/IEC 18000-6 Type A tags through the **AFI** mechanism.

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.

If the tag does not support **AFI** programming (see Annex C.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.

C.6.3 System information: DSFID

The **DSFID** is supported in ISO/IEC 18000-6 Type A tags through the **DSFID** mechanism.

The **DSFID** is returned during the Inventory process. The **DSFID** can be programmed and locked by specific commands. The **DSFID** is defined on 8 bits.

If the tag does not support **DSFID** programming (see Annex C.6.5), the tag shall return **DSFID** = '00'.

C.6.4 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 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.

C.6.5 Support for commands

ISO/IEC 18000-6 Type A 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.

C.6.6 Performance optimisation

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

C.7 Tag Driver for ISO/IEC 18000-6 Type B: 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 RFID tags compliant to ISO/IEC 18000-6 Type B.

C.7.1 Defining the Singulation-Id

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.

The UID format for Type B is specified in ISO/IEC 18000-6 in Clause B.2

C.7.2 System information: AFI

The **AFI** is supported in ISO/IEC 18000-6 Type B tags 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 $0A_{16}$ 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 (see C.7.5), 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.

C.7.3 System information: DSFID

The **DSFID** is supported in ISO/IEC 18000-6 Type B tags through the **DSFID** mechanism.

The **DSFID** indicates how the application data is structured in the tag application memory. As defined in Annex C.7.2, bytes 12 through 17 are reserved in tag system memory for representation of tag memory system information. Byte 12 with the value $0A_{16}$ 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 (see C.7.5), the tag shall return **DSFID** = '00'.

C.7.4 Memory-related parameters

The first 18 bytes (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.

C.7.5 Support for commands

ISO/IEC 18000-6 Type B 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.

C.7.6 Performance optimisation

If the application and/or the interrogator requires a continuous dialogue with a specific tag, ISO/IEC 18000-6 Type B provides a mechanism for "selection" through the command set. The group selection commands (GROUP_SELECT_XX) provide an efficient means to logically query the contents of either system memory (e.g. UID, AFI, 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.

C.8 Tag Driver for ISO/IEC 18000-6 Type C: 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 RFID tags compliant to ISO/IEC 18000-6 Type C.

C.8.1 Defining the Singulation-Id

Each RFID tag is defined uniquely by a dynamic 16-bit code, the RN-16. The value of the RN-16 is returned during selection processes, and is used to maintain communications with the RFID tag while in the operating field. In addition, access to Memory Bank 01 returns a Unique Item Identifier, which does provide a unique identifier at the application level, so long as this is distinguished between EPCglobal code structures and ISO code structures.

C.8.2 System information: AFI

The **AFI** is supported in ISO/IEC 18000-6 Type C as an optional element in the protocol control word in Memory Bank 01. The **AFI** is encoded in bits 18_{16} to $1F_{16}$. To identify that the encoding of the tag is compliant with **AFI** and **DSFID** rules, bit 17_{16} shall have the value = 1, to distinguish this from encoding compliant with the EPCglobal scheme.

The **AFI** only requires to be encoded in Memory Bank 01 to fulfil its function with the RFID tag selection process.

C.8.3 System information: DSFID

The **DSFID** is supported in ISO/IEC 18000-6 Type C RFID tag in the manner described in the following sub-clauses, which may be implemented independently.

C.8.3.1 DSFID in Memory Bank 01

A DSFID is used in Memory Bank 01 when this is called for by an AFI. In this case Memory Bank 01 shall encode the **DSFID** in position 20_{16} to 27_{16}. The encoding that follows shall be compliant with this International Standard for encoding a UII based on an **Object-Identifier** structure.

When the **AFI** declares a **Monomorphic-UII**, the **DSFID** shall not be encoded in Memory Bank 01.

C.8.3.2 DSFID in Memory Bank 11

If Memory Bank 11 is present on the ISO/IEC 18000-6C RFID tag and encodes data based on **Object-Identifier** structures, then the **DSFID** shall be encoded in the first byte position of this memory bank (i.e. bits 00_{16} to 07_{16}).

C.8.3.3 Extensions mechanisms for the Data-Format, Access-Method, and other tag functions

If any of the extension mechanisms defined in 9.2.6 to 9.2.17 are required, then all of the extension bytes shall be encoded starting immediately after the encoded DSFID. All the extension bytes shall precede any encoded packets of data.

C.8.4 Memory-related parameters

The memory of an ISO/IEC 18000-6 Type C RFID tag is organised into four logical sectors known as Memory Bank 00_2, 01_2, 10_2, 11_2. All read and write processes are based on transactions in multiples of 16-bit words. ISO/IEC 18000-6 Amd 1 defines that locking is for an entire Memory Bank.

Memory Bank 00_2 contains passwords that are used to provide permission to access other memory, and to carry out the kill function.

Memory Bank 01_2 contains the Unique Item Identifier, either this is an EPCglobal-based code or an ISO-based code. Processes on the RFID tag calculate the length of the encoding. This length (in words) encoded in bits 00_{16} to 04_{16}, which is part of the Protocol Control word.

Memory Bank 10_2 contains tag identification details, which may be based on a code structure managed by EPCglobal, or a Unique Tag ID compliant with ISO/IEC 15693. There is no simple way to determine whether a tag uses one of the code structures or the other without reading this Memory Bank.

Memory Bank 11_2 contains the encoding of application data beyond the basic Unique Item Identifier and the encoding capacity of Memory Bank 01_2. There are no air interface rules to determine the size of this memory (i.e. the equivalent of the number of words or blocks), without an understanding of the IC manufacturer's specification or looking up information based on the encoded values in Memory Bank 10_2.

C.8.5 Memory Locking

If the tag is complaint with ISO/IEC 18000-6C AMD1 locking is only allowed for the complete memory bank. If the tag is compliant with a version of ISO/IEC 18000-6C later than AMD1 then selective locking of MB11 is an optional feature, which is declared by each tag model.

> NOTE There are no currently defined rules in the air interface protocol to identify whether a particular RFID supports selective locking to the rules defined above. Until such rules are defined this remains as a vendor specific solution for the RFID tag to declare its capability and a vendor specific solution for interrogator manufacturers to implement.

C.8.6 Support for commands

ISO/IEC 18000-6 Type C supports a number of commands including read and write. A number of the commands and associated functions are optional. The response to any command includes specific error codes where the command could not be executed.

C.9 Tag Driver for ISO/IEC 18000-6 Type D: 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 RFID tags compliant to ISO/IEC 18000-6 Type D.

C.9.1 Defining the Singulation-Id

Each tag is identified uniquely by a **Singulation-Id** (also known as TID-S), which can have one of three lengths. The **Singulation-Id** format is specified in ISO/IEC 18000-6 Type D. The TID-S is complaint with ISO/IEC 15963 and can therefore begin with one of the following byte values and having a pre-defined length:

— $E0_{16}$, with a total length of 64 bits

— $E2_{16}$, with a total length of up to 192 bits

— $E3_{16}$, with a total length of 96 bits

Each structure has an internal rule that enables the total lemgth of the TID-S to be self-declaring. Each scheme also supports an IC manufacture plus model number that makes it possible to identify the functions supported by a particular tag.

Each tag returns its **Singulation-Id** during the tag communication process, and is at the beginning of the transmitted message.

C.9.2 System information: AFI

Bit positions 63 to 48 of the first data page shall encode the Protocol Control word. The **AFI** is an element in the Protocol Control word in the UII segment of memory. The **AFI** is encoded in bits 55_{10} to 48_{10}. To identify that the encoding of the tag is compliant with **AFI** and **DSFID** rules, bit 56_{10} shall have the value = 1, to distinguish this.

The **AFI** only requires to be encoded in the UII segment of memory to fulfil its function with the RFID tag selection process.

C.9.3 System information: DSFID

The **DSFID** is supported in an ISO/IEC 18000-6 Type D RFID tag in the manner described in the following sub-clauses, which may be implemented independently.

C.9.3.1 DSFID in UII memory segment

C.9.3.1.1 DSFID when the AFI declares a Monomorphic-UII

When the **AFI** declares a **Monomorphic-UII**, the **DSFID** shall not be encoded in the UII memory segment.

C.9.3.1.2 DSFID for other AFI types

If the UII segment is compliant with encoding rules defined in this International Standard, then in addition to the **AFI,** the UII segment shall encode a single byte **DSFID** in position 47_{10} to 40_{10} of the UII segment, or a multiple byte DSFID starting at position 47_{10}. The encoding that follows shall be compliant with this International Standard for encoding a UII based on an **Object-Identifier** structure.

C.9.3.2 DSFID in the item-related data segment

If the item-related data segment is present on the ISO/IEC 18000-6 Type D RFID tag and encodes data based on **Object-Identifier** structures, then the **DSFID** shall be encoded in the second byte position of this memory bank. This position varies depending on the end point of the UII segment.

The preceding byte is the length-lock byte that

— In the leading five bits encodes the encoded length of the Item-related segment (in words) from the beginning of the length-lock byte to the end of the CRC

— The trailing three bits provide an indication of the pages that are locked

C.9.3.3 Extensions mechanisms for the Data-Format, Access-Method, and other tag functions

If any of the extension mechanisms defined in 9.2.6 to 9.2.17 are required, then all of the extension bytes shall be encoded starting immediately after the encoded **DSFID**. All the extension bytes shall precede any encoded packets of data.

C.9.4 Memory-related parameters

When used in a compliant manner with this International Standard, the memory of an ISO/IEC 18000-6 Type D RFID tag is organised into four logical sectors known as:

— The TID segment

— the UII segment, which begins on the first data page of the ISO/IEC 18000-6 Type D tag

— the item-related data segment (optional)

— the simple sensor encoding segment (optional); always 64 bits if present

All read and write processes are based on transactions in multiples of 16-bit words, organised in 64-bit pages.

C.9.5 Memory Locking

Memory locking is applied in a consistent manner for any one Type D tag, with locking over 16-bits, 32-bits, or 64-bits as specified by the tag manufacturer.

> NOTE There are no currently defined rules in the air interface protocol to identify whether a particular RFID supports selective locking to the rules defined above. Until such rules are defined this remains as a vendor specific solution for the RFID tag to declare its capability and a vendor specific solution for interrogator manufacturers to implement.

C.9.5.1 Encoding and locking the UII segment

The UII segment begins immediately after the locked TID-S segment, and therefore begins on a 64-bit page boundary.

The UII segment may end on any 16-bit word, unless there is an application requirement for it to be locked. If the UII segment requires locking and the UII does not end on a lock boundary, then pad lock bytes (value 00_{16}) are included until the end of the lock boundary.

C.9.5.2 Encoding and locking the Item-related data segment

The optional item-related data segment is encoded if the Protocol Control word bit 58_{10} is set = 1. This segment begins immediately after the end of the UII segment, and therefore begins on a 16-bit word boundary, which does not have to be on a page boundary.

The first byte (bit positions n to {n-7} of the item-related data segment is the length-lock indicator, which provides precise information about the number of words in the segment and basic information on whether pages in the segment are locked.

The second byte (bit positions {n-8} to {n-15} of the item-related data segment is the DSFID for this segment, which may be different from the DSFID of the UII segment.

The item-related data is encoded from bit {n-16} of the item-related data segment until the end of the segment, using the rules of this International Standard. Data sets within the encoded sequence may be selectively locked. Any data set, or contiguous data sets that are to be locked, shall be on lock boundaries.

C.9.6 Support for commands

ISO/IEC 18000-6 Type D communicates its data payload without the need for air interface commands. As currently specified, there are no encoding commands defined. Therefore encoding is achieve in a vendor specific manner across the air interface, but which need to support the encoding as defined by the functional write commands as described in Annex A.4.

Annex D
(normative)

Encoding rules for No-Directory Access-Method

This Annex, and associated referenced annexes, specifies the encoding rules for the **Access-Method No-Directory**, which is the basic encoding scheme for this International Standard. The process described in the following sub-clauses builds up the structure of a **Data-Set** in a logical process: starting with the compacting the **Object**, encoding the length of the compacted **Object**, processing the **Object-Identifier**, formatting this into a **Relative-OID** wherever possible, encoding the length of the **Object-Identifier** or the **Relative-OID**, and encoding the Precursor and Offset byte (if required).

D.1 Object processing

Data compaction is applied as a transformation to the data **Objects** to reduce the number of bits that are transferred across the air interface and that are required to encode data in memory. The compaction shall be done according to the **Compact-Parameter** (defined in 10.1.9) received in the ISO/IEC 15961 application commands. Only **Compact-Parameters** 0, 1, 2 apply to the encoding of this **Access-Method**, and **Compact-Parameters** 0, 2, and 15 apply to the decoding.

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 RFID 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.

Data compaction performs all the processes necessary to compact a data **Object** and to determine the Compaction Type, which is encoded on the RFID tag as part of the Precursor. The **Object-Identifier** remains unchanged and is not subject to any form of compaction, to enable it to be directly identifiable by the application and the Logical Memory. The command argument **Object-Lock** remains unchanged and is passed through to the next stage of processing: data formatting.

D.1.1 Compaction process

The command argument **Compact-Parameter** determines whether the **Object** is subject to the compaction process or not, based on the following integer values.

0 **Application-Defined:** The data object is read by and passed through the data compaction process without any compaction being applied, but is assigned the Compaction Type Code 000_2.

1 **Compact:** The data object is read by and passed through the data compaction process to be compacted as efficiently as possible and assigned the appropriate Compaction Type Code in the range 001_2 to 110_2.

2 **UTF8-Data:** The data object is read by and passed through the data compaction process without any compaction being applied, but is assigned the Compaction Type Code 111_2 to indicate that it is compliant with the UTF-8 transformation of ISO/IEC 10646.

The **Object** is read by and passed through the data compaction process.

1. The data **Object** itself is transformed to its compacted form. If the command argument **Compact-Parameter:**

   a. is set to 1 (**Compact**) the data **Object** input string is compacted (see Annex D.1.2 and Annex E for the detailed processes).

b. is set to 0 (**Application-Defined**), or to 2 (**UTF8-Data**) 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 Annex D.1.3).

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

D.1.2 Compaction schemes

Data compaction shall be applied to each entire data **Object**. When compaction is requested by the **Compact-Parameter**, 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 D.1 — Determining the Data Compaction Scheme, 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 E**.**

Table D.1 — 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) $\neq 30_{16}$ 2. Length of object > 1 byte 3. Length of object ≤ 19 bytes	integer	E.1
30 to 39	Length of object > 1 byte	numeric	E.2
41 to 5F	Length of object > 2 bytes	5 bit code	E.3
20 to 5F	Length of object > 3 bytes	6 bit code	E.4
00 to 7E	Length of object > 7 bytes	7 bit code	E.5
00 to FF	N/A	octet string	E.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 F) 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 **Object-Identifier** given externally in the application.

D.1.3 Compaction Type codes

The codes for the **Compaction Type** are defined in Table D.2 — Compaction Type Codes. One of these is applied to the data **Object** after it has been through the data compaction process. On subsequent reading from the RFID tag, this is used to identify how the data object shall be de-compacted.

Table D.2 — 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

D.2 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 and encoded 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 the last byte = 0 and the first bit of all intervening bytes = 1.
      1bbbbbbb 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.

   NOTE Although probably an unrealistic requirement the rule (in step 3.a) can be extended to cover any length.

D.3 Processing the Object-Identifier

The **Object-Identifier** in the application commands is compliant with the rules of ISO/IEC 8824-1 and may be presented in a style compliant with that standard (see 6.3.1) or as a Uniform Resource Name (see 6.3.3). It may also be presented in a truncated form as a **Relative-OID**, and this structure is discussed in Annex D.4.

The **Object-Identifier** value is encoded as a series of byte aligned values as follows:

1. The first two arcs of the registration tree are encoded as a single integer using the formula:

   40f + s

where f = the value of the first arc
s = the value of the second arc

2. The value "n" of each additional arc is encoded into a byte-aligned-bit-field. This is done as follows for values of "n":

   a. For n < 128:

      the decimal value is converted to binary and encoded in a single byte; thus bit 8 is set to zero

   b. For : 128 ≤ n < 16384:

      the decimal value is converted to binary and subdivided into two 7-bit strings: bit 7 to bit 1, bit 14 to bit 8. Each of these new bit strings is encoded in a byte; with bit 8 of the first byte set to 1, bit 8 of the last byte set to 0.

   c. For n ≥ 16384:

      the decimal value is converted to binary and subdivided into 7-bit strings: bit 7 to bit 1, bit 14 to bit 8, bit 21 to bit 15, and so on. Each of these new bit strings is encoded in a byte; with bit 8 of the first byte set to 1, bit 8 of the last byte set to 0 and bit 8 of the intervening byte(s) set to 1. The example below shows the process.

EXAMPLE

1. Value = 91234_{10} = 1 01100100 01100010_2
2. Split into 7-bit strings

   0000101 1001000 1100010
3. Add prefix bits 0 for last byte

   1 for preceding byte(s)

   10000101 **1**1001000 **0**1100010

Using this technique, the length of each component arc of the **Object-Identifier** is self declaring. The first byte always defines the first two arcs. Each subsequent arc is defined by one byte if the lead bit of that byte is 0; and multiple bytes if the lead bit is 1, the group of bytes ends with the byte with its lead bit equal 0. The arc value is encoded in the sequence of 7-bit values.

EXAMPLE

[00101000]	1[1111000]	0[1001010]	0[0000001]
(1 x 40) + 0	15434		1
1 0	15434		1

Although the number of arcs allows for an **Object-Identifier** of any length, this International Standard limits the length of the encoded value to be no more than 127 bytes. This is a constraint placed to meet encoding requirements on the RFID tag and the structure of the Logical Memory.

NOTE The constraint is on the encoded length of the **Object-Identifier** and not the number of arcs. It should also be understood that an **Object-Identifier** encoded in 127 bytes is highly unlikely.

D.3.1 Encoding the Object-Identifier

The need to encode the full **Object-Identifier** is rare, because the formatting rules enable it to be truncated in most cases. In those situations where the **Object-Identifier** has to be encoded, then the structure, as described above, represents the bytes that need to be encoded.

The **Object-Identifier** is preceded by a length code as defined in Annex D.5.

D.3.2 The Data-Set structure when the Object-Identifier is encoded

In the situations where a full **Object-Identifier** needs to be encoded, the **Data-Set** is structured as defined in Table D.3 — Data-Set structures for Object-Identifiers, which clearly identifies that the length of the **Object-Identifier** has to be calculated and precede the encoded value of the **Object-Identifier**.

Table D.3 — Data-Set structures for Object-Identifiers

Description	**Structure of Byte String for an Encoded Data Set**						
Object-Identifier	Precursor	Length of Object-Identifier	Object-Identifier ~~	Length of data	Data ~~		
Object-Identifier with Offset	Precursor	Offset	Length of Object-Identifier	Object-Identifier	Length of data	Data ~~	Pad ~~

NOTE ~~ indicates that this component typically can be multiple bytes.

The number of pad bytes could be 0 – 254, depending on the block size and lock block alignment

D.4 Processing the Relative-OID

In some circumstances, a truncated form of the **Object-Identifier**, the **Relative-OID**, can be incorporated into application commands.

As the first arc(s) of the **Object-Identifier** is declared by some other means (e.g. the **Data-Format**), the remaining arcs that are presented shall be encoded according to the rules of Step 2 in the encoding rules defined in Annex D.3.

In some cases, the application will only provide the single final arc and, for most practical examples, this is likely to be a value either less than 128 or less than 16384.

D.4.1 Formatting the Object-Identifier based on the Data-Format

The **Data-Format** provides a shorthand method to achieve efficient encoding of the **Object-Identifier**. This is done by the **Data-Format** declaring a specific common **Root-OID** that is used to truncate the full **Object-Identifier** into a **Relative-OID**. For encoding, the byte values representing the **Root-OID** are deleted from the beginning of the full **Object-Identifier** to create the **Relative-OID**. For decoding, The **Root-OID** is concatenated as a prefix to the **Relative-OID** to re-create an **Object-Identifier** compliant with ISO/IEC 8824-1.

The processes described below are based on a typical write application command being processed by the Data Processor for encoding on an RFID tag. Where there are significant variants these are described.

A. If **Data-Format = Not-Formatted (0), Closed-System (30),** or **Extension (31)**

An error has occurred and formatting is not possible.

B. If **Data-Format = Full-Featured (1)**

Each **Object-Identifier** shall be encoded as a full **Object-Identifier**, as defined in Annex D.3.1. If a **Relative-OID** is presented, the application command is in error and formatting is not possible

C. If **Data-Format =** **Root-OID-Encoded (2)**

1. Parse the set of **Object-Identifier** values to identify any common **Root-OID** that has at least two arcs. If the RFID tag already has an encoded **Root-OID**, establish whether this can be applied to the new **Object-Identifiers**. Once a common **Root-OID** has been identified, any other **Object-Identifier** 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 RFID tag.

3. Strip out the common **Root-OID** from each **Object-Identifier** 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 **Object-Identifier** 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 G, subsequently processed with a **Data-Format = Root-OID-Encoded** is illustrated in Annex G.5.1.

D. If **Data-Format =** Any code value {3 .. 29}, as defined in the Register of data constructs to the rules of ISO/IEC 15961-2, with some examples illustrated in Table D.4 — Example mappings of Data-Format to Root-OID.

Table D.4 — Example mappings of Data-Format to Root-OID

Data-Format	Application	common Root-OID
3	ISO15434	{1 0 15434} {28 F8 4A $_{16}$}
4	ISO6523	{1 0 6523} {28 B2 7B $_{16}$}
5	ISO15459	{1 0 15459} {28 F8 63 $_{16}$}
6	Library-Loan-Items	{1 0 15961 8} {28 FC 59 08 $_{16}$}
8	ISO15961-Combined	{1 0 15961} {28 FC 59 $_{16}$}
10	Data-Identifier-Algorithm	{1 0 15961 10} {28 FC 59 0A $_{16}$}
12	IATA-Baggage	{1 0 15961 12} {28 FC 59 0C $_{16}$}

NOTE: The example of the library items clearly indicates that there is not requirement for a one-to-one relationship between the Data-Format and the final arc of the Root-OID

1. Parse the set of **Object-Identifier** values to identify the common **Root-OID**, as defined on the register of data constructs (with illustrated examples in Table D.4 — Example mappings of Data-Format to Root-OID).

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 **Object-Identifier** to create a **Relative-OID** from the remaining arcs.

   NOTE The **Root-OID** is not encoded, but implied by the **Data-Format**

4. Each other **Object-Identifier** 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 G, subsequently processed with a **Data-format** = **Data-Identifier-Algorithm** is illustrated in Annex G.5.2.

D.4.2 Encoding the Root-OID for Data-Format = Root-OID-Encoded (2)

A **Root-OID** up to 126 bytes long can be encoded. The value of the length is directly encoded in the Precursor (see Annex D.6.2), 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 Annex D.4.1. 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.

D.4.3 The Data-Set structure when the Root-OID is encoded

In the situations where a **Root-OID** needs to be encoded, the **Data-Set** is structured as defined in Table D.5 — Data-Set structures for Root-OID. This identifies that a length value 00_{16} shall be encoded, to enable the beginning of the next **Data-Set** to be identified If the Offset is required for block alignment, then the appropriate number of Pad bytes shall be encoded.

Table D.5 — Data-Set structures for Root-OID

Description	Structure of Byte String for an Encoded Data Set						
Root-OID	Precursor	Length of Root-OID	Root-OID ~~	Length of data = 00_{16}			
Root-OID with Offset	Precursor	Offset	Length of Root-OID	Root-OID ~~	Length of data = 00_{16}	Pad ~~	

NOTES: ~~ indicates that this component typically can be multiple bytes.

The number of pad bytes could be 0 – 254, depending on the block size and lock block alignment

D.4.4 Encoding the Relative-OID

If the **Relative-OID** is provided by the application, then it is already in a form for encoding in the rules defined in this sub-clause. If the application provides the full **Object-Identifier**, then the Relative-OID needs to be derived using the truncation rules described in Annex D.4.1.

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

D.4.4.1 Single arc Relative-OID value 1 to 14

The final byte of the Object-Identifier (representing the single arc value 1 to 14) transferred from the application interface has a value 01 to $0E_{16}$. This single arc value shall be encoded in bits 4 to 1 of the Precursor as defined in Annex D.6. No additional bytes are encoded in the Logical Memory for this Object-Identifier value and its length.

D.4.4.2 Single arc Relative-OID value 15 to 127

The final arc of the **Object-Identifier** (representing the single arc value 15 to 127) transferred from the application interface ha a value 0F to $7F_{16}$. The single arc shall be encoded in a single byte as follows:

$$0bbbbbbb_2$$

Where bbbbbbb = arc value -15

EXAMPLE

RELATIVE-OID	=	23
bbbbbbb	=	23 – 15 = 8
	=	00001000_2
	=	08_{16}

No additional bytes are required to encode the length of the Relative-OID with this range of values.

D.4.4.3 Other Relative-OID values

Irrespective of the number of arcs and the value of those arcs, other **Relative-OID** values shall be encoded with the byte values presented in the application command or as derived from processing the **Data-Format** to strip off the **Root-OID**.

The **Relative-OID** is preceded by a length code as defined in Annex D.5.

D.4.5 The Data-Set structure when the Relative-OID is encoded

In the situations where a **Relative-OID** needs to be encoded, the **Data-Set** is structured as defined in Table D.6 — Data-Set structures for Relative-OIDs. This identifies that if the **Relative-OID** is other than a single arc with the value 1 to 127, then the length of the OID has to be calculated and precede the encoded value of the **Relative-OID**.

Table D.6 — Data-Set structures for Relative-OIDs

Description	Structure of Byte String for an Encoded Data Set						
Single Relative-OID 1 -14	Precursor	Length of data	Data ~~				
Single Relative-OID 1 –14 with Offset	Precursor	Offset	Length of data	Data ~~	Pad ~~		
Single Relative-OID 15 -127	Precursor	Relative-OID	Length of data	Data ~~			
Single Relative-OID 15 –127 with Offset	Precursor	Offset	Relative-OID	Length of data	Data ~~	Pad ~~	
Other Relative-OID	Precursor	Length of OID	OID ~~	Length of data	Data ~~		
Other Relative-OID with Offset	Precursor	Offset	Length of OID	OID ~~	Length of data	Data ~~	Pad ~~

NOTES: ~~ indicates that this component typically can be multiple bytes

The number of pad bytes could be 0 – 254, depending on the block size and lock block alignment

D.5 Encoding the length and Object-Identifier or Relative-OID

The process to encode the length of the **Object-Identifier** or **Relative-OID** is as follows:

1. Record the byte string that represents the **Object-Identifier** or **Relative-OID** value.

2. Count the number of bytes in the **Object-Identifier** or **Relative-OID**.

3. If the length 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 **Object-Identifier** or **Relative-OID** in the remaining 5 bits:
      length + 1 = bbbbb
   d. Encode the **Object-Identifier** or **Relative-OID** in the subsequent bytes.

   NOTE This procedure allows a **Relative-OID** with a single arc value 0 (presented as 00_{16}) to be encoded.

4. If the length 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) in bits 7 to 1 of the second byte.
      10100000 0bbbbbbb
   d. Encode the **Object-Identifier** or **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) = 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39_{16}

Encoding = **A0 1D** 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39_{16}

The first two bytes define the length, but all the other bytes are as provided by the application command, or the process to strip off the **Root-OID** based on the **Data-Format**.

D.6 The Precursor

The Precursor is a single metadata byte that is always the first byte of the Data-Set and provides information about:

- The Compaction Scheme applied to the encoded **Object**.
- The Object-Identifier.
- Whether any block alignment has been necessary (e.g. for locked data)

There are also some values of the Precursor that are used for different functions, as described in some of the sub-clauses below.

D.6.1 The Precursor for Data-Format not equal 2

The Precursor is a single byte that positioned at the beginning of each **Data-Set**. The structure of the Precursor is bit based from bit 8 to bit 1, as follows:

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 Annex D.1.3)

Bits 4 to 1: used to directly encode some **Relative-OID** values

values 0001 to 1110_2 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

D.6.2 The Precursor for the Root-OID for Data-Format = 2

The **Data-Format = Root-OID-Encoded (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.

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 **Data-Format =2** shall be as that defined in Annex D.6.1.

D.6.3 The Precursor value 00_{16} as a terminator

The Precursor with the value 00_{16} acts as a terminator of the **Data-Set** list of **No-Directory** structure. It also acts as a terminator of the directory in a **Directory** structure on the Logical Memory.

D.6.4 The Precursor value 80_{16} for the Null-Byte

The Precursor with the value 80_{16} signals a Null-Byte that is inserted in a sequence of **Data-Sets** when one has been modified or deleted.

In the case of the **Modify** command (see 8.4.3), if the encoded length of the modified **Data-Set** is longer than the previously encoded **Data-Set**, there can be a requirement to re-organise all subsequent data on the RFID tag. An alternative solution is to encode the required number of Null-Bytes in the existing space, and encode the modified **Data-Set** at the end of all existing **Data-Sets**. This particular solution of using the Null-Byte, might not be possible in the situation where the modified **Data-Set**, placed at the end of the encoding, causes the encoding to overflow the memory capacity.

A similar option is available for the Delete command, which simply requires the deleted **Data-Set** to be overwritten with a series of Null-Bytes. The Null-Byte should not be used as the last **Data-Set**, because this is consuming encoding capacity for no logical advantage.

During the decoding process, if the byte value 80_{16} occurs as a valid Precursor, then all contiguously encoded bytes with the same value shall be ignored. The first byte that has a different value shall be processed as the Precursor of the next **Data-Set**.

D.7 The Offset byte

The Offset byte is required if the block(s) containing the **Data-Set** needs to be block aligned. This could be necessary because the particular **Data-Set** or an adjacent **Data-Set** is required to be locked.

The Offset byte shall encode the value of the number of pad bytes (byte value 80_{16}) 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 9.2.2), and the Precursor plus offset require two bytes, the maximum value for the Offset is FE_{16}.

The Offset value 00_{16} 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_{16} shall be used to indicate that the Offset is followed by a Precursor expansion byte (see Annex D.8).

D.8 The Precursor expansion byte

If the Offset byte has the value FF_{16}, 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 or revised, 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_{16} 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.

D.9 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.

D.9.1 Overall decode strategy

The **DSFID** is the prime key to decoding. Particular values determine whether a **Directory** structure is used, which 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 **No-Directory** structure requires continual processing of the **Data-Set** area until the desired **Object-Identifier** is found.

The **DSFID** 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 **No-Directory** structure, or a particular **Data-Set**.

If the **Object-Identifier** 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 **Object-Identifier**. In this case, the first byte(s) of the encoded **Object-Identifier** need to be parsed to determine the length and type of **Object-Identifier**. These points are more fully described below.

D.9.2 Decoding the DSFID

The following **DSFID** values are in the format m, n where m = **Access-Method** (0-1) and n = **Data-Format** (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 -- **No-Directory**, with each **Object-Identifier** presented in full

0,2 -- **No-Directory** with an explicitly encoded **Root-OID** in the first logical position. All subsequent **Relative-OID** values are prefixed by the **Root-OID** when decoded. Full **Object-Identifiers** may appear in the encodation.

0,n (n = 3…287) -- **No-Directory** with the **Root-OID** defined by the **Data-Format.** All subsequent **Relative-OID** values are prefixed by the **Root-OID** when decoded. Full **Object-Identifiers** may appear in the encodation.

1,1 -- **Directory** with each **Object-Identifier** presented as in full 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** when decoded. Full **Object-Identifiers** may appear in the encodation.

1,n (n = 3…287) -- **Directory** with the **Root-OID** defined by the **DSFID**. All subsequent **Relative-OID** values are prefixed by the **Root-OID** when decoded. Full **Object-Identifiers** may appear in the encodation.

NOTE These rules do not necessarily apply to different **Access-Methods** and the detailed specification of other **Access-Methods** should be used as the correct reference.

D.9.3 Decoding the Precursor

The following bit-based structure applies to the **Precursor**, with three 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 Annex D.8) which may be followed by a second offset byte.
7 to 5		Defines the Compaction Type Code (see Annex D.1.3). 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 **Object-Identifier** is treated as a Null field. Value 1111 indicates that the **Precursor** is followed by an encoded **Object-Identifier** (of any permitted type).

D.9.3.1 The Terminator byte

If the **Precursor** has the value 00_{16} it acts as a Terminator to signal the end of the **Directory** and/or the end of the encoding of the **Data-Sets**.

D.9.3.2 The Null-Byte

If the **Precursor** has the value 80_{16} it signals that this and contiguous bytes with the same value have been used to deal with the encoding space following a delete or modify command. The first byte that has a different value shall be treated as the valid **Precursor** of the next **Data-Set**.

D.9.3.3 The Precursor for the explicitly encoded Root-OID

If the **DSFID** 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 Annex D.8) which may be followed by a second offset byte.
7 to 1		Defines the length (in bytes) of the **Root-OID**.

If the **DSFID** 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 **DSFID**, 0,2 and 1,2 shall comply with the decode process defined in xxx.

D.9.4 Decoding the leading byte(s) of the encoded Object-Identifier

Generally, the encoding of the **Object-Identifier** immediately follows the **Precursor**. The only exceptions to this are when the **Precursor** itself encodes the **Object-Identifier** (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 **Object-Identifier** identify the type of **Object-Identifier** 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. To calculate the value of the arc for the application interface add $0F_{16}$ 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 Annex D.5) 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 xxxx), 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 xxxx) 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 xxx) 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**.

Annex E
(normative)

Basic Data Compaction Schemes

The basic data compaction schemes were first introduced in the 2004 version of this International Standard for the **No-Directory Access-Method**. They are also called out in the **Packed-Objects** and **Tag-Data-Profile Access-Methods**. Annexes associated with the other **Access-Methods** make specific reference to sub-clauses below.

The basic data compaction schemes are applicable to data **Objects** to reduce the amount of encoding space required to store that data on the RFID tag. The schemes shall apply to entire data **Object**, 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. If a character string is too short to compact under any one compaction scheme, the character string shall be encoded using octet encodation.

E.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_{16}; and the leading byte(s) shall not be 30_{16}.

If the decimal integer value is less than 10, use octet encodation. If the decimal integer is longer than 19 digits, or the leading byte(s) are 30_{16}, 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 are 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_{16} to achieve the minimum encoded length. The encoded byte string should not include Encode as integer**,** code value 001 in the Precursor.

E.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_{16}.

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 a 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.

E.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 $5F_{16}$. The character string shall be 3 or more characters long. Up to 37% of memory space can be saved using this scheme. Annex F 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 $5F_{16}$.
    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_{16}, which is not supported in 5-bit compaction and shall be discarded.

E.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 $5F_{16}$. If the trailing byte(s) are 20_{16}, 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 F shows the ISO/IEC 646 characters that can be encoded.

The rules for 6-bit compaction are:

1. Check for byte 20_{16} 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 $5F_{16}$.
    b. Convert the byte value to its 8-bit binary equivalent.
    c. Strip off the leading 2 bits: “00” for bytes 20 to $3F_{16}$ or “01” for bytes 40 to $5F_{16}$.
    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 $3F_{16}$.

b. If the first bit is “0”, the bits “01” are added as a prefix before converting to values 40 to $5F_{16}$.

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_{16} 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 compaction process.

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 $5F_{16}$, 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

E.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 $7E_{16}$. The character string shall be 8 or more characters long. Up to 12% of memory space can be saved using this scheme. Annex F 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 $7E_{16}$.
    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.

EXAMPLE

The **Object** content {Ace#123451337} converts to HEX as 41 63 65 23 31 32 33 34 35 31 33 33 37 Analysing this byte stream shows that all values are in the range 00 to $7E_{16}$, enabling 7-bit compaction to be used. The Object byte stream converts as follows:

HEX: 41 63 65 23 31 32 33 34 35 31 33 33 37

Binary: 01000001 01100011 01100101 00100011 00110001 00110010 00110011 00110100 00110101 00110001 00110011 00110011 00110111

Remove bit 8: 1000001 1100011 1100101 0100011 0110001 0110010 0110011 0110100 0110101 0110001 0110011 0110011 0110111

As this is only 91 bits, the first five bits of the pad string "11111" are appended and the 96 bit string is divided into a sequence of 8-bit values:

10000011 10001111 00101010 00110110 00101100 10011001 10110100 01101010 11000101 10011011 00110110 11111111

Convert to HEX: 83 8F 2A 36 2C 99 B4 6A C5 9B 36 FF

NOTE Although the last encoded byte contains all 1s, decoding from the first byte in 7 bit steps ensures that the pad bits are correctly recognised and discarded

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 $7F_{16}$, which is not supported in 7-bit compaction and shall be discarded.

EXAMPLE

The encoded string { AF CB 0E EC FB 32 F2 40 BE 0C 28 71 22 FF } is presented to the decoder, and processed as follows:

Convert to 8-bits: 10101111 11001011 00001110 11101100 11111011 00110010 11110010 01000000 10111110 00001100 00101000 01110001 00100010 11111111

Concatenate & split to 7-bit: 1010111 1110010 1100001 1101110 1100111 1101100 1100101 1110010 0100000 0101111 1000001 1000010 1000011 1000100 1000101 1111111

Add lead 0 to each 7-bits:	01010111 01110010 01100001 01101110 01100111 01101100 01100101 01110010 00100000 00101111 01000001 01000010 01000011 01000100 01000101 01111111
Convert to HEX:	57 72 61 6E 67 6C 65 72 20 2F 41 42 43 44 45 7F
Convert to ASCII:	**W r a n g l e r / A B C D E**

NOTE The HEX byte **7F** at the end of the encoding appears to be a valid character, except that **7F** is not a supported character for the 7-bit compaction scheme. The apparent ambiguity is caused by the fact that the 15 character string {Wrangler /ABCDE} encodes over 105 bits and requires 7 pad bits at the end to encode on a byte aligned basis. During the decode process the 14 encoded bytes (112 bits) are concatenated and split into sixteen 7-bit sequences each of which is prefixed by a leading 0 bit. The last 7-bit sequence {1111111} indicates a pad sequence. When this has the 0 prefix this appears as a legitimate byte, except for the fact that **7F** is an invalid character, and so is discarded. This occurs for character strings of 15, 23, 31 and so forth in length.

E.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 F
(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	•	•		
%	25	•	•		
&	26	•	•		
‘	27	•	•		
(	28	•	•		
)	29	•	•		
*	2A	•	•		

ISO/IEC 646 Character	Octet Value (HEX)	Included in Compaction Type			
		7-bit	6-bit	5-bit	Numeric
+	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	•	•	•	
Q	51	•	•	•	
R	52	•	•	•	
S	53	•	•	•	
T	54	•	•	•	
U	55	•	•	•	
V	56	•	•	•	
W	57	•	•	•	
X	58	•	•	•	
Y	59	•	•	•	
Z	5A	•	•	•	
[	5B	•	•	•	
\	5C	•	•	•	

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

Annex G
(informative)

Encoding example for No-Directory structure

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. Specific implementations are likely to be more efficient, but the detailed steps are shown to aid readers with an understanding of the process.

G.1 Starting position

This stage is based on the transfer of the **Object-Identifiers** and **Objects** in application commands. This example uses two hypothetical **Object-Identifiers** and **Objects,** but not considering any additional arguments such as locking the **Data-Set**.

EXAMPLE

Using the urn form (see 6.3.3) for the **Object-Identifier** and printable characters for the **Object**, the input values are:

urn:oid: 1.0.15691.27.48 with the nine character data “ABC123456”

urn:oid:1.0.15961.27.13 with the two character data “50”

The data is probably transferred as the hexadecimal strings {41 42 43 31 32 33 34 35 36} and {35 30}, respectively.

G.2 Encoding the Object-Identifiers

For this illustration, assume that the full **Object-Identifier** is first created in the correct format before any truncation to remove the common **Root-OID**. This is a procedure that could be used with **Data-Format** = 2, but for other **Data-Formats** it is possible to identify the **Root-OID** and only encode the **Relative-OID**.

Using the rules defined in Annex D.3,

urn:oid: 1.0.15691.27.48 encodes as {28 FC 59 1B 30}

urn:oid:1.0.15961.27.13 encodes as {28 FC 59 1B 0D}

G.3 The initial state of the entry for the Logical Memory

Using the example in Annex G.2, the bytes can be mapped to create the initial encoding as illustrated in Table G.1 — Initial Encoding Example of Application Data. The **Precursor** bits are set to NULL. All the other encoding is of bytes.

NOTE The contents of Table G.1 — Initial Encoding Example of Application Data will be updated as additional processes are applied.

Table G.1 — Initial Encoding Example of Application Data

Precursor (bits)								Offset	L of Object-Id	Object-Identifier	L of Object	Object
b8	b7	b6	b5	b4	b3	b2	b1					
									05	28 FC 59 1B 30	09	41 42 43 31 32 33 34 35 36
									05	28 FC 59 1B 0D	02	35 30

G.4 The Logical Memory after data compaction

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

The compaction process (as defined in Annex D.1 and Annex E) can be applied to the two data objects in Table G.1 — Initial Encoding Example of Application Data.

— 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 E.4 for the detailed conversion).

— The data **Object** 35 30 is compacted to 1 byte {32} using the integer Type (see Annex E.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

The results are shown in Table G.2 — Encoding example after data compaction.

Table G.2 — Encoding example after data compaction

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

G.5 The Logical Memory after formatting for a No-Directory Access-Method

The format rules have a different effect depending on the **Data-Format** values. The following sub-clauses illustrate this using the same input with different **Data-Formats**.

G.5.1 The Logical Memory after formatting for Data-Format = 2 (Root-OID-Encoded)

Table G.3 — Encoding example for Data-Format = 2 shows the result of formatting the input of Table G.2 — Encoding example after data compaction with **Data-Format = 2 (Root-OID-Encoded):**

— The Root-OID 28 FC 59 1B 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 **Object-Identifier** 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_{16}

— 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 G.3 — Encoding example for Data-Format = 2

Precursor (bits)								Offset	Object-Id Type & Length	Object-Identifier	L of Object	Object
b8	b7	b6	b5	b4	b3	b2	b1					
	0	0	0	0	1	0	0			28 FC 59 1B	00	
	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

G.5.2 The Logical Memory after formatting for Data-Format not equal 2

Table G.4 — Encoding Example of Data-Format not equal 2 shows the result of formatting the input of Table G.2 — Encoding example after data compaction with **Data-Format** not equal 2. This clearly shows that the **Root-OID** is not encoded, because it is directly indicated by the particular **Data-Format**. In all other respects, the encodation is as in Table G.3 — Encoding example for Data-Format = 2, because the same procedures are used to create the precursor for each **Object-Identifier** / **Object** pair.

Table G.4 — Encoding Example of Data-Format not equal 2

Precursor (bits)								Offset	Object-Id Type & Length	Object-Identifier	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 H
(informative)

Encoding example for Directory structure

The following example illustrates the principles to construct a **Directory**.

H.1 The base data

Table H.1 — Example encoded byes without a Directory shows 64 bytes of encoding for 9 **Data-Sets**, with the shaded cells indicating compacted data. The byte values in white cells therefore represent the Precursor, the encoding of the **Object-Identifier** and length bytes.

Table H.1 — Example encoded byes without a Directory

		2nd nibble of HEX Address															
		x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**xA**	**xB**	**xC**	**xD**	**xE**	**xF**
1st nibble of HEX Address	**0x**	4F	82	32	07	04	20	F1	CB	3D	35	DA	1D	01	32	11	06
	1x	0B	3A	73	CE	2F	F2	02	01	F8	43	0A	38	CB	71	CB	3D
	2x	35	DB	7E	39	C2	44	05	38	CB	79	E7	98	15	02	04	C0
	3x	16	05	02	4C	B0	16	EA	17	06	70	48	86	0D	DF	79	00

H.2 Encoding the first Directory entry

Each **Directory** entry takes leading bytes of the **Data-Set** as follows:

— The Precursor for the **Data-Set**.

— The length of the **Object-Identifier** (conditional).

— The **Object-Identifier** (conditional)

As the first **Data-Set** has the **Relative-OID** value 48_{10}, each of these components needs to be encoded. In hexadecimal, the byte sequence is:

$4F_{16}$ = the Precursor
82_{16} = the length of the **Object-Identifier**
30_{16} = the **Object-Identifier**

The address of the Precursor for this **Data-Set** is **00_{16}**. This address applies irrespective of the block size of the Logical Memory. The complete sequence is encoded, starting from the highest addressed block in the memory. For this example assume that the block size is 8 bytes. The bytes are encoded in the highest block, but still with the normal read direction as shown in Table H.2 — The start of the Directory.

Table H.2 — The start of the Directory

		2nd nibble (HEX) of block							
		x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**
1st nibble (HEX) of block	**(n-3)x**								
	(n-2)x								
	(n-1)x								
	(n)x	4F	82	32	**00**				

H.3 Encoding the second Directory entry

The second Data-Set has its Relative-OID encoded in the Precursor, so only the value (**$1D_{16}$**) of the Precursor is encoded in the Directory. This is followed by the byte address (**$1D_{16}$**) of the Precursor. These bytes are encoded immediately after the previous **Directory** entry, as shown in Table H.3 — The second Directory entry.

Table H.3 — The second Directory entry

		2nd nibble (HEX) of block							
		x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**
1st nibble (HEX) of block	**(n-3)x**								
	(n-2)x								
	(n-1)x								
	(n)x	4F	82	32	**00**	1D	**0B**		

H.4 Encoding the remaining Directory entries

The remaining Data-Sets are processed to add the **Object-Identifier** and location to the **Directory**, as follows:

— The third **Data-Set** only requires the Precursor and its location to be encoded, and this is entered at the end of the highest block (based on this example of an 8 byte block).

— The fourth **Data-Set** only requires the Precursor and its location to be encoded, and this is entered at the beginning of the next highest block (based on this example of an 8 byte block).

— The fifth **Data-Set** only requires the Precursor and its location to be encoded, and this is entered in the block (n-1), based on this example of an 8 byte block.

— The sixth **Data-Set** only requires the Precursor and its location to be encoded, and this is entered in the block (n-1), based on this example of an 8 byte block.

— The seventh **Data-Set** only requires the Precursor and its location to be encoded, and this is entered in the block (n-1), based on this example of an 8 byte block.

— The eighth **Data-Set** only requires the Precursor and its location to be encoded, and this is entered in the block (n-2), based on this example of an 8 byte block.

— The ninth **Data-Set** only requires the Precursor and its location to be encoded, and this is entered in the block (n-1), based on this example of an 8 byte block.

— The Terminator is encoded in the next byte location of the **Directory**.

The complete encoded directory is shown in Table H.4 — The complete Directory entries, and is encoded in blocks (n-2) to block n, where "n" is the value of the highest block.

Table H.4 — The complete Directory entries

		2nd nibble (HEX) of block							
		x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**
1st nibble (HEX) of block	**(n-3)x**								
	(n-2)x	16	**30**	17	**37**	00			
	(n-1)x	02	**17**	43	**1A**	44	**26**	15	**2C**
	(n)x	4F	82	32	**00**	1D	**0B**	11	**0E**

In this example, all the directory blocks are located at higher block address than the encoded data, otherwise it is not possible to encode a **Directory**. As block (n-3) has not been used for the **Directory**, subsequent encoding of data or directory mat be placed in this block.

H.5 Decoding the Directory and reading the target Object-Identifier

The **Directory** is decoded from the highest block address. The first byte is the Precursor of the first **Data-Set** and the structure to the end of the **Object-identifier** is self-declaring and is decoded as defined in Annex D.9.3 and D.9.4. As the location address of the target Precursor is encoded to the length encoding rules defined in Annex D.2, it too has a self-declaring structure. If the location is below position 128, 1 byte is required; for locations at addresses up to 16383, two bytes are required; and additional bytes can be used for any size of address.

With these structuring rules it is possible to parse and decode the **Directory** until the target **Object-Identifier** and its location are found.

Using the block size, it is possible to identify the block containing the Precursor of the target **Object-Identifier** and to invoke a selective read transaction across the air interface.

Annex I
(normative)

Packed-Objects structure

I.1 Overview

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

I.2 Overview of associated Annexes

The formal description of Packed Objects is presented in this Annex and in the annexes defined as follows:

— The overall structure of Packed Objects is described in Annex I.3

— The individual sections of a Packed Object are described in Annex I.4 to I.9

— The structure and features of ID Tables (utilized by Packed Objects to represent various data system identifiers) are described in Annex J.

— The numerical bases and character sets used in Packed Objects are described in Annex K.

— An encoding algorithm and worked example are described in Annex L.

— The decoding algorithm for Packed Objects is described in Annex M.

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

I.3 High-level Packed-Objects format design

I.3.1 Overview

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

NOTE Because the data structures within an encoded Packed object are bit-aligned rather than byte-aligned, this Annex uses the term 'octet' instead of 'byte' except in cases where an eight-bit quantity must be aligned on a byte boundary.

DSFID[1]	Optional Pointer[2] And/Or Padding	First Packed-Object	Optional Pointer[2] And/Or Padding	Optional Second Packed-Object	...	Optional Packed-Object	Optional Pointer[2] And/Or Padding	Zero Octet(s)

Note: 1. For many tag architectures the **DSFID** is encoded separately in a predefined area of memory, and not as illustrated here.
2. The Optional Pointer to a Directory Packed Object may appear at most only once in memory.

Figure I.1 — Overall memory structure when using Packed-Objects

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

— An ID List Packed Object (IDLPO) encodes a series of ID Values as a list, whose length depends on the number of data items being represented;

— An ID Map Packed Object (IDMPO) instead encodes a fixed-length bit array, whose length depends on the total number of entries defined in the registered Base Table. Each bit in the array is '1' if the corresponding table entry is represented by the Packed Object, and is '0' otherwise.

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

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

Figure I.2 — Packed Object Structure

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

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

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

A Packed Object may optionally begin with the pattern '0000', which is reserved to introduce one or more **Format Flags**, as described in Annex I.4.2. These flags may indicate use of the non-default ID Map format. If the Format Flags are not present, then the Packed Object defaults to the ID List format:

— Certain flag patterns indicate an inter-Object pattern (Directory Pointer or Padding)

— Other flag patterns indicate the Packed Object's type (Map or List), and may indicated the presence of an optional Addendum subsection that enables data to be edited.

All Packed Objects contain an **Object Info Section** that includes Object Length Information and ID Value Information:

— Object Length Information includes an ObjectLength field (indicating the overall length of the Packed Object in octets) followed by Pad Indicator bit, so that the number of significant bits in the Packed Object can be determined.

— ID Value Information indicates which Identifiers are present and in what order, and (if an IDLPO) also includes a leading NumberOfIDs field, indicating how many ID Values are encoded in the ID List.

The Object Info section is encoded in one of the following formats, as shown in Figure I.3 — IDLPO Object Info Structure and Figure I.4 — IDMPO Object Info Structure.

— ID List (IDLPO) Object Info format:

- Object Length (EBV-6) plus Pad Indicator bit
- A single ID List or an ID Lists Section (depending on Format Flags)

— ID Map (IDMPO) Object Info format:

- One or more ID Map sections
- Object Length (EBV-6) plus Pad Indicator bit

For either of these Object Info formats, an Optional Addendum subsection may be present at the end of the Object Info section.

Object Info, in a Default ID List PO			
Object Length	Number Of IDs	ID List	Optional Addendum

or

Object Info, in a Non-default ID List PO		
Object Length	ID Lists Section (one or more lists)	Optional Addendum

Figure I.3 — IDLPO Object Info Structure

Object Info, in an ID Map PO		
ID Map Section (one or more maps)	Object Length	Optional Addendum

Figure I.4 — IDMPO Object Info Structure

A Packed Object may include a **Secondary ID Section**, if needed to encode additional bits that are defined for some classes of IDs (these bits complete the definition of the ID).

A Data Packed Object may include an **Aux Format Section**, which if present encodes one or more bits that are defined to support data compression, but do not contribute to defining the ID

A Data Packed Object includes a **Data Section**, representing the compressed data associated with each of the identifiers listed within the Packed Object. This section is omitted in a Directory Packed Object, and in a Packed Object that uses **No-directory** compaction (see I.7.1). Depending on the declaration of data format in the relevant ID table, the Data section will contain either or both of two subsections:

— **Known-Length Numerics subsection:** this subsection compacts and concatenates all of the non-empty data strings that are known a priori to be numeric.

— **AlphaNumeric subsection:** this subsection concatenates and compacts all of the non-empty data strings that are not a priori known to be all-numeric.

I.4 Format Flags section

This sub-clause defines the valid Format Flags patterns that may appear at the expected start of a Packed Object to override the default layout if desired. For example, a Format Flag can change the Packed Object's format, or signal the insertion of padding patterns to align the next Packed Object on a word or block boundary). The set of defined patterns is shown in Table I.1 — Format Flags.

Table I.1 — Format Flags

Bit Pattern	Description	Additional Info	See Annex
0000 0000	Termination Pattern	No more packed objects follow	I.4.1
0000 0001	Invalid Pattern		
0000 0010	Invalid Pattern		
0000 0011	Invalid Pattern		
LLLLLL xx	First octet of an IDLPO	For any LLLLLL > 3	I.5
0000	Format Flags starting pattern	(if the full EBV-6 is non-zero)	I.4.2
0000 10NA	IDLPO with: N = 1: non-default Info A = 1: Addendum Present	If N = 1: allows multiple ID tables If A = 1: Addendum pointer(s) at end of Object Info section	I.4.3
0000 01xx	Inter-Packed Object pattern	A Directory Pointer, or padding	I.4.4
0000 0100	Signifies a padding byte	No padding length indicator follows	I.4.4
0000 0101	Signifies run-length padding	An EBV-8 padding length follows	I.4.4
0000 0110	RFU		I.4.4
0000 0111	Directory pointer	Followed by EBV-8 pattern	I.4.4
0000 11xx	ID Map Packed Object		I.4.2

I.4.1 Data Terminating Flag Pattern

A pattern of eight or more '0' bits at the expected start of a Packed Object denotes that no more Packed Objects are present in the remainder of memory.

NOTE Six successive '0' bits at the expected start of a Packed Object would (if interpreted as a Packed Object) indicate an ID List Packed Object of length zero.

I.4.2 Format Flags section starting bit patterns

A non-zero EBV-6 with a leading pattern of "0000" is used as a Format Flags section Indicating Pattern. The additional bits following an initial '0000' Format Flag Indicating Pattern are defined as follows:

— A following two-bit pattern of '10' (creating an initial pattern of '000010') indicates an IDLPO with at least one non-default optional feature (see Annex I.4.3)

— A following two-bit pattern of '11' indicates an IDMPO, which is a Packed Object using an ID Map format instead of ID List-format The ID Map section (see Annex I.9) immediately follows this two-bit pattern.

— A following two-bit pattern of '01' signifies an External pattern (Padding pattern or Pointer) prior to the start of the next Packed Object (see Annex I.4.4).

A leading EBV-6 ObjectLength of less than four is invalid as a Packed Objects length.

NOTE The shortest possible Packed Object is an IDLPO, for a data system using four bits per ID Value, encoding a single ID Value. This Packed Object has a total of 14 fixed bits. Therefore, a two-octet Packed Object would only contain two data bits, and is invalid. A three-octet Packed Object would be able to encode a single data item up to three digits long. In order to preserve "3" as an invalid length in this scenario, the Packed Objects encoder shall encode a leading Format Flags section (with all options set to zero, if desired) in order to increase the object length to four.

I.4.3 IDLPO Format Flags

The presence of the bit pattern '000010' at the expected start of a Packed Object is followed by two additional bits, to form a complete IDLPO Format Flag of "000010NA", where:

— If the first additional bit 'N' is '0', then the default IDLPO format is employed for the IDLPO Object Info section This allows for only a single ID List (utilizing the registration's default Base ID Table).

— If the first additional bit 'N' is '1', then a non-default format is employed for the IDLPO Object Info section. The optional non-default IDLPO Object Info format supports a sequence of one or more ID Lists, and each such list begins with identifying information as to which registered table it represents (see Annex I.5.1).

— If the second additional bit 'A' is '1', then an Addendum subsection is present at the end of the Object Info section (see Annex I.5.6).

— If the second additional bit 'A' is '0', then no Addendum subsection is present.

I.4.4 Patterns for use between Packed Objects

The presence of '000001' at the expected start of a Packed Object is used to indicate either padding or a directory pointer, as follows:

A following two-bit pattern of '11' indicates that a Directory Packed Object Pointer follows the pattern. The pointer is one or more octets in length, in EBV-8 format. This pointer may be Null (a value of zero), but if non-zero, indicates the number of octets from the start of the pointer to the start of a Directory Packed Object (which if editable, shall be the first in its "chain"). For example, if the Format Flags byte for a Directory Pointer is encoded at byte offset 1, the Pointer itself occupies bytes beginning at offset 2, and the Directory starts at byte offset 9, then the Dir Ptr encodes the value "7" in EBV-8 format. A Directory Packed Object Pointer may appear before the first Packed Object in memory, or at any other position where a Packed Object may begin, but may only appear once in a given data carrier memory, and (if non-null) must be at a lower address than the Directory it points to. The first octet after this pointer may be padding (as defined immediately below), a new set of Format Flag patterns, or the start of an ID List Packed Object.

— A following two-bit pattern of '00' indicates that the full eight-bit pattern of '00000100' pattern serves as a padding byte, so that the next Packed Object may begin on a desired word or block boundary. This pattern may repeat as necessary to achieve the desired alignment.

— A following two-bit pattern of '01' indicates as a run-length padding indicator, and shall be immediately followed by an EBV-8 indicating the number of octets from the start of the EBV-8 itself to the start of the next Packed Object (for example, if the next Packed Object follows immediately, the EBV-8 has a value of one). This mechanism eliminates the need to write many words of memory in order to pad out a large memory block.

— A following two-bit pattern of '10' is Reserved.

I.5 Object Info section

Each Packed Object's Object Info section contains both Length Information (the size of the Packed Object, in bits and in octets), and ID Values Information. A Packed Object encodes representations of one or more data system Identifiers and (if a Data Packed Object) also encodes their associated data elements (Application Identifier (AI) strings, Data Identifier (DI) strings, etc). The ID Values information encodes a complete listing of all the Identifiers (AIs, DIs, etc) encoded in the Packed Object, or (in a Directory Packed Object) all the Identifiers encoded anywhere in memory.

To conserve encoded and transmitted bits, data system Identifiers (each typically represented in data systems by either two, three, or four ASCII characters) is represented within a Packed Object by an ID Value, representing an index denoting an entry in a registered Base Table of ID Values. A single ID Value may represent a single Object Identifier, or may represent a commonly-used sequence of Object Identifiers. In some cases, the ID Value represents a "class" of related Object Identifiers, or an Object Identifier sequence in which one or more Object Identifiers are optionally encoded; in these cases, Secondary ID Bits (see Annex I.6) are encoded in order to specify which selection or option was chosen when the Packed Object was encoded. A "fully-qualified ID Value" (FQIDV) is an ID Value, plus a particular choice of associated Secondary ID bits (if any are invoked by the ID Value's table entry). Only one instance of a particular fully-qualified ID Value may appear in a data carrier's Data Packed Objects, but a particular ID Value may appear more than once, if each time it is "qualified" by different Secondary ID Bits. If an ID Value does appear more than once, all occurrences shall be in a single Packed Object (or within a single "chain" of a Packed Object plus its Addenda).

There are two methods defined for encoding ID Values: an ID List Packed Object uses a variable-length list of ID Value bit fields, whereas an ID Map Packed Object uses a fixed-length bit array. Unless a Packed Object's format is modified by an initial Format Flags pattern, the Packed Object's format defaults to that of an ID List Packed Object (IDLPO), containing a single ID List, whose ID Values correspond to the default Base ID Table of the registered Data Format. Optional Format Flags can change the format of the ID Section to either an IDMPO format, or to an IDLPO format encoding an ID Lists section (which supports multiple ID Tables, including non-default data systems).

Although the ordering of information within the Object Info section varies with the chosen format (see Annex I.5.1), the Object Info section of every Packed Object shall provide Length information as defined in Annex I.5.2, and ID Values information (see Annex I.5.3) as defined in Annex I.5.4, or I.5.5. The Object Info section (of either an IDLPO or an IDMPO) may conclude with an optional Addendum subsection (see Annex I.5.6.

I.5.1 Object Info formats

I.5.1.1 IDLPO default Object Info format

The default IDLPO Object Info format is used for a Packed Object either without a leading Format Flags section, or with a Format Flags section indicating an IDLPO with a possible Addendum and a default Object Info section. The default IDLPO Object Info section contains a single ID List (optionally followed by an Addendum subsection if so indicated by the Format Flags). The format of the default IDLPO Object Info section is shown in Table I.2 — Default IDLPO Object Info format.

Table I.2 — Default IDLPO Object Info format

Field Name:	Length Information	NumberOfIDs	ID Listing	Addendum subsection
Usage:	The number of octets in this Object, plus a last-octet pad indicator	Number of ID Values in this Object (minus one)	A single list of ID Values; value size depends on registered Data Format	Optional pointer(s) to other Objects containing Edit information
Structure:	Variable: see Annex I.5.2	Variable:EBV-3	See Annex I.5.4	See Annex I.5.6.1

In a IDLPO's Object Info section, the NumberOfIDs field is an EBV-3 Extensible Bit Vector, consisting of one or more repetitions of an Extension Bit followed by 2 value bits. This EBV-3 encodes one less than the number of ID Values on the associated ID Listing. For example, an EBV-3 of '101 000' indicates (4 + 0 +1) = 5 IDs values. The Length Information is as described in Annex I.5.2 for all Packed Objects. The next fields are an ID Listing (see Annex I.5.4) and an optional Addendum subsection (see Annex I.5.6).

I.5.1.2 IDLPO non-default Object Info format

Leading Format Flags may modify the Object Info structure of an IDLPO, so that it may contain more than one ID Listing, in an ID Lists section (which also allows non-default ID tables to be employed). The non-default IDLPO Object Info structure is shown in Table I.3 — Non-Default IDLPO Object Info format.

Table I.3 — Non-Default IDLPO Object Info format

Field Name:	Length Info	ID Lists section, first List			Optional ID List section(s)	Null App Indicator (single zero bit)	Addendum Subsection
		Application Indicator	Number of IDs	ID Listing			
Usage:	The number of octets in this Object, plus a last-octet pad indicator	Indicates the selected ID Table and the size of each entry	Number Of ID Values on the list	Listing of ID Values, then one F/R Use bit	Repeated lists, each for a different ID Table	Zero or more repeated lists, each for a different ID Table, followed by a null Application Indicator (single zero bit)	Optional pointer(s) to other Objects containing Edit information
Structure:	See Annex I.5.2	See Annex I.5.3.1	See Annex I.5.1.1	See Annex I.5.4 and I.5.3.2	References in previous columns	See Annex I.5.3.1	See Annex I.5.6

I.5.1.3 IDMPO Object Info format

Leading Format Flags may define the Object Info structure to be an IDMPO, in which the Length Information (and optional Addendum subsection) follow an ID Map section (see Annex I.5.5). This arrangement ensures that the ID Map is in a fixed location for a given application, of benefit when used as a Directory. The IDMPO Object Info structure is shown in Table I.4 — IDMPO Object Info format.

Table I.4 — IDMPO Object Info format

Field Name:	ID Map section	Length Information	Addendum
Usage:	One or more ID Map structures, each using a different ID Table	The number of octets in this Object, plus a last-octet pad indicator	Optional pointer(s) to other Objects containing Edit information
Structure:	See Annex I.9.1	See Annex I.5.2	See Annex I.5.6

I.5.2 Length Information

The format of the Length information, always present in the Object Info section of any Packed Object, is shown in Table I.5 — Packed Object Length information.

Table I.5 — Packed Object Length information

Field Name:	ObjectLength	Pad Indicator
Usage:	The number of 8-bit bytes in this Object. This includes the 1st byte of this Packed Object, including its IDLPO/IDMPO format flags if present. It excludes patterns for use between packed objects, as specified in I.4.4.	If '1': the Object's last byte contains at least 1 pad bit
Structure:	Variable: EBV-6	Fixed: 1 bit

The first field, ObjectLength, is an EBV-6 Extensible Bit Vector, consisting of one or more repetitions of an Extension Bit and 5 value bits. An EBV-6 of '000100' (value of 4) indicates a four-byte Packed Object. An EBV-6 of '100001 000000' (value of 32) indicates a 32-byte Object, and so on.

The Pad Indicator bit immediately follows the end of the EBV-6 ObjectLength. This bit is set to '0' if there are no padding bits in the last byte of the Packed Object. If set to '1', then bitwise padding begins with the least-significant or rightmost '1' bit of the last byte, and the padding consists of this rightmost '1' bit, plus any '0' bits to the right of that bit. This method effectively uses a *single* bit to indicate a *three*-bit quantity (i.e., the number of trailing pad bits). When a receiving system wants to determine the total number of bits (rather than bytes) in a Packed Object, it would examine the ObjectLength field of the Packed Object (to determine the number of bytes) and multiply the result by eight, and (if the Pad Indicator bit is set) examine the last byte of the Packed Object and decrement the bit count by (1 plus the number of '0' bits following the rightmost '1' bit of that final byte).

I.5.3 General description of ID Values

A registered data format defines (at a minimum) a Primary Base ID Table (a detailed specification for registered ID tables may be found in Annex J). This base table defines the data system Identifier(s) represented by each row of the table, any Secondary ID Bits or Aux Format bits invoked by each table entry, and various implicit rules (taken from a predefined rule set) that decoding systems shall use when interpreting data encoded according to each entry. When a data item is encoded in a Packed Object, its associated table entry is identified by the entry's relative position in the Base Table. This table position or index is the ID Value that is represented in Packed Objects.

A Base Table containing a given number of entries inherently specifies the number of bits needed to encode a table index (i.e., an ID Value) in an ID List Packed Object (as the Log (base 2) of the number of entries). Since current and future data system ID Tables will vary in unpredictable ways in terms of their numbers of table entries, there is a need to pre-define an ID Value Size mechanism that allows for future extensibility to accommodate new tables, while minimizing decoder complexity and minimizing the need to upgrade decoding software (other than the addition of new tables). Therefore, regardless of the exact number of Base Table entries defined, each Base Table definition shall utilize one of the predefined sizes for ID Value encodings defined in Table I.6 — Defined ID Value sizes (any unused entries shall be labeled as reserved, as provided in Annex J). The ID Size Bit pattern is encoded in a Packed Object only when it uses a non-default Base ID Table. Some entries in the table indicate a size that is not an integral power of two. When encoding (into an IDLPO) ID Values from tables that utilize such sizes, each pair of ID Values is encoded by multiplying the earlier ID of the pair by the base specified in the fourth column of Table I.6 — Defined ID Value sizes and adding the later ID of the pair, and encoding the result in the number of bits specified in the fourth column. If there is a trailing single ID Value for this ID Table, it is encoded in the number of bits specified in the third column of Table I.6 — Defined ID Value sizes.

Table I.6 — Defined ID Value sizes

ID Size Bit pattern	Maximum number of Table Entries	Number of Bits per single or trailing ID Value, and how encoded	Number of Bits per pair of ID Values, and how encoded
000	Up to 16	4, as 1 Base 16 value	8, as 2 Base 16 values
001	Up to 22	5, as 1 Base 22 value	9, as 2 Base 22 values
010	Up to 32	5, as 1 Base 32 value	10, as 2 Base 32 values
011	Up to 45	6, as 1 Base 45 value	11, as 2 Base 45 values
100	Up to 64	6, as 1 Base 64 value	12, as 2 Base 64 values
101	Up to 90	7, as 1 Base 90 value	13, as 2 Base 90 values
110	Up to 128	7, as 1 Base 128 value	14, as 2 Base 128 values
1110	Up to 256	8, as 1 Base 256 value	16, as 2 Base 256 values
111100	Up to 512	9, as 1 Base 512 value	18, as 2 Base 512 values
111101	Up to 1024	10, as 1 Base 1024 value	20, as 2 Base 1024 values
111110	Up to 2048	11, as 1 Base 2048 value	22, as 2 Base 2048 values
111111	Up to 4096	12, as 1 Base 4096 value	24, as 2 Base 4096 values

I.5.3.1 Application Indicator subsection

An Application Indicator subsection can be utilised to indicate use of ID Values from a default or non-default ID Table. This subsection is required in every IDMPO, but is only required in an IDLPO that uses the non-default format supporting multiple ID Lists.

An Application Indicator consists of the following components:

— A single AppIndicatorPresent bit, which if '0' means that no additional ID List or Map follows. Note that this bit is always omitted for the first List or Map in an Object Info section. When this bit is present and '0', then none of the following bit fields are encoded.

— A single ExternalReg bit that, if '1', indicates use of an ID Table from a registration other than the memory's default. If '1', this bit is immediately followed by a 9-bit representation of a Data Format registered under ISO/IEC 15961.

— An ID Size pattern which denotes a table size (and therefore an ID Map bit length, when used in an IDMPO), which shall be one of the patterns defined by Table I.6 — Defined ID Value sizes. The table size indicated in this field must be less than or equal to the table size indicated in the selected ID table. The purpose of this field is so that the decoder can parse past the ID List or ID Map, even if the ID Table is not available to the decoder.

— A three-bit ID Subset pattern. The registered data format's Primary Base ID Table, if used by the current Packed Object, shall always be indicated by an encoded ID Subset pattern of '000'. However, up to seven Alternate Base Tables may also be defined in the registration (with varying ID Sizes), and a choice from among these can be indicated by the encoded Subset pattern. This feature can be useful to define smaller sector-specific or application-specific subsets of a full data system, thus substantially reducing the size of the encoded ID Map.

I.5.3.2 Full/Restricted Use bits

When contemplating the use of new ID Table registrations, or registrations for external data systems, application designers may utilise a "restricted use" encoding option that adds some overhead to a Packed Object but in exchange results in a format that can be fully decoded by receiving systems not in possession of the new or external ID table. With the exception of a IDLPO using the default Object Info format, one Full/Restricted Use bit is encoded immediately after each ID table is represented in the ID Map section or ID Lists section of a Data or Directory Packed Object. In a Directory Packed object, this bit shall always be set to '0' and its value ignored. If an encoder wishes to utilise the "restricted use" option in an IDLPO, it shall preface the IDLPO with a Format Flags section invoking the non-default Object Info format.

If a “Full/Restricted Use” bit is ‘0’ then the encoding of data strings from the corresponding registered ID Table makes full use of the ID Table’s IDstring and FormatString information. If the bit is ‘1’, then this signifies that some encoding overhead was added to the Secondary ID section and (in the case of Packed-Object compaction) the Aux Format section, so that a decoder without access to the table can nonetheless output OIDs and data from the Packed Object according to the scheme specified in Annex J.4.1. Specifically, a Full/Restricted Use bit set to ‘1’ indicates that:

— for each encoded ID Value, the encoder added an EBV-3 indicator to the Secondary ID section, to indicate how many Secondary ID bits were invoked by that ID Value. If the EBV-3 is nonzero, then the Secondary ID bits (as indicated by the table entry) immediately follow, followed in turn by another EBV-3, until the entire list of ID Values has been represented.

— the encoder did not take advantage of the information from the referenced table’s FormatString column. Instead, corresponding to each ID Value, the encoder inserted an EBV-3 into the Aux Format section, indicating the number of discrete data string lengths invoked by the ID Value (which could be more than one due to combinations and/or optional components), followed by the indicated number of string lengths, each length encoded as though there were no FormatString in the ID table. All data items were encoded in the A/N subsection of the Data section.

I.5.4 ID Values representation, in an ID List Packed Object

Each ID Value is represented within an IDLPO on a list of bit fields; the number of bit fields on the list is determined from the NumberOfIDs field (see Table I.2 — Default IDLPO Object Info format). Each ID Value bit field’s length is in the range of four to eleven bits, depending on the size of the Base Table index it represents. In the optional non-default format for an IDLPO’s Object Info section, a single Packed Object may contain multiple ID List subsections, each referencing a different ID Table. In this non-default format, each ID List subsection consists of an Application Indicator subsection (which terminates the ID Lists, if it begins with a ‘0’ bit), followed by an EBV-3 NumberOfIDs, an ID List, and a Full/Restricted Use flag.

I.5.5 ID Values representation, in an ID Map Packed Object

Encoding an ID Map can be more efficient than encoding a list of ID Values, when representing a relatively large number of ID Values (constituting more than about 10 percent of a large Base Table’s entries, or about 25 percent of a small Base Table’s entries). When encoded in an ID Map, each ID Value is represented by its relative position within the map (for example, the first ID Map bit represents ID Value “0”, the third bit represents ID Value “2”, and the last bit represents ID Value ‘n’ (corresponding to the last entry of a Base Table with (n+1) entries). The value of each bit within an ID Map indicates whether the corresponding ID Value is present (if the bit is ‘1’) or absent (if ‘0’). An ID Map is always encoded as part of an ID Map Section structure (see Annex I.9.1).

I.5.6 Optional Addendum subsection of the Object Info section

The Packed Object Addendum feature supports basic editing operations, specifically the ability to add, delete, or replace individual data items in a previously-written Packed Object, without a need to rewrite the entire Packed Object. A Packed Object that does not contain an Addendum subsection cannot be edited in this fashion, and must be completely rewritten if changes are required.

An Addendum subsection consists of a Reverse Links bit, followed by a Child bit, followed by either one or two EBV-6 links. Links from a Data Packed Object shall only go to other Data Packed Objects as addenda; links from a Directory Packed Object shall only go to other Directory Packed Objects as addenda. The standard Packed Object structure rules apply, with some restrictions that are described in Annex I.5.6.2.

The Reverse Links bit shall be set identically in every Packed Object of the same “chain.” The Reverse Links bit is defined as follows:

— If the Reverse Links bit is ‘0’, then each child in this chain of Packed Objects is at a higher memory location then its parent. The link to a Child is encoded as the number of octets (plus one) that are in between the last octet of the current Packed Object and the first octet of the Child. The link to the

parent is encoded as the number of octets (plus one) that are in between the first octet of the parent Packed Object and the first octet of the current Packed Object.

— If the Reverse Links bit is '1', then each child in this chain of Packed Objects is at a lower memory location than its parent. The link to a Child is encoded as the number of octets (plus one) that are in between the first octet of the current Packed Object and the first octet of the Child. The link to the parent is encoded as the number of octets (plus one) that are in between the last octet of the current Packed Object and the first octet of the parent.

The Child bit is defined as follows:

— If the Child bit is a '0', then this Packed Object is an editable "Parentless" Packed Object (i.e., the first of a chain), and in this case the Child bit is immediately followed by a single EBV-6 link to the first "child" Packed Object that contains editing addenda for the parent.

— If the Child bit is a '1', then this Packed Object is an editable "child" of an edited "parent," and the bit is immediately followed by one EBV-6 link to the "parent" and a second EBV-6 line to the next "child" Packed Object that contains editing addenda for the parent.

A link value of zero is a Null pointer (no child exists), and in a Packed Object whose Child bit is '0', this indicates that the Packed Object is editable, but has not yet been edited. A link to the Parent is provided, so that a Directory may indicate the presence and location of an ID Value in an Addendum Packed Object, while still providing an interrogator with the ability to efficiently locate the other ID Values that are logically associated with the original "parent" Packed Object. A link value of zero is invalid as a pointer towards a Parent.

In order to allow room for a sufficiently-large link, when the future location of the next "child" is unknown at the time the parent is encoded, it is permissible to use the "redundant" form of the EBV-6 (for example using "100000 000000" to represent a link value of zero).

I.5.6.1 Addendum "EditingOP" list (only in ID List Packed Objects)

In an IDLPO only, each Addendum section of a "child" ID List Packed Object contains a set of "EditingOp" bits encoded immediately after its last EBV-6 link. The number of such bits is determined from the number of entries on the Addendum Packed Object's ID list. For each ID Value on this list, the corresponding EditingOp bit or bits are defined as follows:

— '1' means that the corresponding Fully-Qualified ID Value (FQIDV) is Replaced. A Replace operation has the effect that the data originally associated with the FQIDV matching the FQIDV in this Addendum Packed Object shall be ignored, and logically replaced by the Aux Format bits and data encoded in this Addendum Packed Object)

— '00' means that the corresponding FQIDV is Deleted but not replaced. In this case, neither the Aux Format bits nor the data associated with this ID Value are encoded in the Addendum Packed Object.

— '01' means that the corresponding FQIDV is Added (either this FQIDV was not previously encoded, or it was previously deleted without replacement). In this case, the associated Aux Format Bits and data shall be encoded in the Addendum Packed Object.

NOTE If an application requests several "edit" operations at once (including some Delete or Replace operations as well as Adds) then implementations can achieve more efficient encoding if the Adds share the Addendum overhead, rather than being implemented in a new Packed Object.

I.5.6.2 Packed Objects containing an Addendum subsection

A Packed Object containing an Addendum subsection is otherwise identical in structure to other Packed Objects. However, the following observations apply:

— A “parentless” Packed Object (the first in a chain) may be either an ID List Packed Object or an ID Map Packed Object (and a parentless IDMPO may be either a Data or Directory IDMPO). When a “parentless” PO is a directory, only directory IDMPOs may be used as addenda. A Directory IDMPO’s Map bits shall be updated to correctly reflect the end state of the chain of additions and deletions to the memory bank; an Addendum to the Directory is not utilized to perform this maintenance (a Directory Addendum may only add new structural components, as described later in this section). In contrast, when the edited parentless object is an ID List Packed Object or ID Map Packed Object, its ID List or ID Map cannot be updated to reflect the end state of the aggregate Object (parents plus children).

— Although a “child” may be either an ID List or an ID Map Packed Object, only an IDLPO can indicate deletions or changes to the current set of fully-qualified ID Values and associated data that is embodied in the chain.

  — When a child is an IDMPO, it shall only be utilized to add (not delete or modify) structural information, and shall not be used to modify existing information. In a Directory chain, a child IDMPO may add new ID tables, or may add a new AuxMap section or subsections, or may extend an existing PO Index Table or ObjectOffsets list. In a Data chain, an IDMPO shall not be used as an Addendum, except to add new ID Tables.

  — When a child is an IDLPO, its ID list (followed by “EditingOp” bits) lists only those FQIDVs that have been deleted, added, or replaced, relative to the cumulative ID list from the prior Objects linked to it.

I.6 Secondary ID Bits section

The Packed Objects design requirements include a requirement that all of the data system Identifiers (AI’s, DI’s, etc.) encoded in a Packed Object’s can be fully recognised without expanding the compressed data, even though some ID Values provide only a partially-qualified Identifier. As a result, if any of the ID Values invoke Secondary ID bits, the Object Info section shall be followed by a Secondary ID Bits section. Examples include a four-bit field to identify the third digit of a group of related Logistics AIs.

Secondary ID bits can be invoked for several reasons, as needed in order to fully specify Identifiers. For example, a single ID Table entry’s ID Value may specify a choice between two similar identifiers (requiring one encoded bit to select one of the two IDs at the time of encoding), or may specify a combination of required and optional identifiers (requiring one encoded bit to enable or disable each option). The available mechanisms are described in Annex J. All resulting Secondary ID bit fields are concatenated in this Secondary ID Bits section, in the same order as the ID Values that invoked them were listed within the Packed Object. Note that the Secondary ID Bits section is identically defined, whether the Packed Object is an IDLPO or an IDMPO, but is not present in a Directory IDMPO.

I.7 Aux Format section

The Aux Format section of a Data Packed Object encodes auxiliary information for the decoding process. A Directory Packed Object does not contain an Aux Format section. In a Data Packed Object, the Aux Format section begins with “Compact-Parameter” bits as defined in Table I.7 — Compact-Parameter bit patterns.

Table I.7 — Compact-Parameter bit patterns

Bit Pattern	Compaction method used in this Packed Object	Reference
‘1’	“Packed-Object” compaction	Annex I.7.2
‘000’	“Application-Defined”, as defined for the No-Directory access method	Annex I.7.1
‘001’	“Compact”, as defined for the No-Directory access method	Annex I.7.1
‘010’	“UTF-8”, as defined for the No-Directory access method	Annex I.7.1
‘011bbbb’	(‘bbbb’ shall be in the range of 4..14): reserved for future definition	Annex I.7.1

If the Compact-Parameter bit pattern is ‘1’, then the remainder of the Aux Format section is encoded as described in Annex I.7.2; otherwise, the remainder of the Aux Format section is encoded as described in Annex I.7.1.

I.7.1 Support for No-Directory compaction methods

If any of the No-Directory compaction methods were selected by the Compact-Parameter bits, then the Compact-Parameter bits are followed by a byte-alignment padding pattern consisting of zero or more ‘0’ bits followed by a single ‘1’ bit, so that the next bit after the ‘1’ is aligned as the most-significant bit of the next byte.

This next byte is defined as the first octet of a “No-Directory Data section”, which is used in place of the Data section described in Annex I.8. The data strings of this Packed Object are encoded in the order indicated by the Object Info section of the Packed Object, compacted exactly as described in Annex D (Encoding rules for No-Directory Access-Method), with the following two exceptions:

— The Object-Identifier is not encoded in the “No-Directory Data section”, because it has already been encoded into the Object Info and Secondary ID sections.

— The Precursor is modified in that only the three Compaction Type Code bits are significant, and the other bits in the Precursor are set to ‘0’.

Therefore, each of the data strings invoked by the ID Table entry are separately encoded in a modified data set structure as:

<modified precursor> <length of compacted object> <compacted object octets>

The <compacted object octets> are determined and encoded as described in Annex D.1.1 and D.1.2. and the <length of compacted object> is determined and encoded as described in D.2.

Following the last data set, a terminating precursor value of zero shall not be encoded (the decoding system recognises the end of the data using the encoded ObjectLength of the Packed Object).

I.7.2 Support for the Packed-Object compaction method

If the Packed-Object compaction method was selected by the Compact-Parameter bits, then the Compact-Parameter bits are followed by zero or more Aux Format bits, as may be invoked by the ID Table entries used in this Packed Object. The Aux Format bits are then immediately followed by a Data section that uses the Packed-Object compaction method described in I.8.

An ID Table entry that was designed for use with the Packed-Object compaction method can call for various types of auxiliary information beyond the complete indication of the ID itself (such as bit fields to indicate a variable data length, to aid the data compaction process). All such bit fields are concatenated in this portion, in the order called for by the ID List or Map. Note that the Aux Format section is identically defined, whether the Packed Object is an IDLPO or an IDMPO.

An ID Table entry invokes Aux Format length bits for all entries that are not specified as fixed-length in the table (however, these length bits are not actually encoded if they correspond to the last data item encoded in the A/N subsection of a Packed Object). This information allows the decoding system to parse the decoded data into strings of the appropriate lengths. An encoded Aux Format length entry utilizes a variable number of bits, determined from the specified range between the shortest and longest data strings allowed for the data item, as follows:

— If a maximum length is specified, and the specified range (defined as the maximum length minus the minimum length) is less than eight, or greater than 44, then lengths in this range are encoded in the fewest number of bits that can express lengths within that range, and an encoded value of zero represents the minimum length specified in the format string. For example, if the range is specified as from three to six characters, then lengths are encoded using two bits, and ‘00’ represents a length of three.

— Otherwise (including the case of an unspecified maximum length), the value (actual length – specified minimum) is encoded in a variable number of bits, as follows:

  — Values from 0 to 14 (representing lengths from 1 to 15, if the specified minimum length is one character, for example) are encoded in four bits

  — Values from 15 to 29 are encoded in eight bits (a prefix of '1111' followed by four bits representing values from 15 ('0000') to 29 ('1110')

  — Values from 30 to 44 are encoded in twelve bits (a prefix of '1111 1111' followed by four bits representing values from 30 ('0000') to 44 ('1110')

  — Values greater than 44 are encoded as a twelve-bit prefix of all '1's, followed by an EBV-6 indication of (value – 44).

NOTES 1 If a range is specified with identical upper and lower bounds (i.e., a range of zero), this is treated as a fixed length, not a variable length, and no Aux Format bits are invoked.

2 If a range is unspecified, or has unspecified upper or lower bounds, then this is treated as a default lower bound of one, and/or an unlimited upper bound.

I.8 Data section

A Data section is always present in a Packed Object, except in the case of a Directory Packed Object or Directory Addendum Packed Object (which encode no data elements), the case of a Data Addendum Packed Object containing only Delete operations, and the case of a Packed Object that uses No-directory compaction (see I.7.1). When a Data section is present, it follows the Object Info section (and the Secondary ID and Aux Format sections, if present). Depending on the characteristics of the encoded IDs and data strings, the Data section may include one or both of two subsections in the following order: a Known-Length Numerics subsection, and an AlphaNumerics subsection. The following paragraphs provide detailed descriptions of each of these Data Section subsections. If all of the subsections of the Data section are utilised in a Packed Object, then the layout of the Data section is as shown in Figure I.5 — Maximum Structure of a Packed Objects Data section.

Known-Length Numeric subsection				AlphaNumeric subsection							
				A/N Header Bits				Binary Data Segments			
1st KLN Binary	2nd KLN Binary	…	Last KLN Binary	Non-Num Base Bit(s)	Prefix Bit, Prefix Run(s)	Suffix Bit, Suffix Run(s)	Char Map	Ext'd. Num Binary	Ext'd Non-Num Binary	Base 10 Binary	Non-Num Binary

Figure I.5 — Maximum Structure of a Packed Objects Data section

I.8.1 Known-length-Numerics subsection of the Data Section

For always-numeric data strings, the ID table may indicate a fixed number of digits (this fixed-length information is not encoded in the Packed Object) and/or a variable number of digits (in which case the string's length was encoded in the Aux Format section, as described above). When a single data item is specified in the FormatString column (see AnnexI.8J.2.3) as containing a fixed-length numeric string followed by a variable-length alphanumeric string, the numeric string is encoded in the Known-length-numerics subsection and the alphanumeric string in the alphanumeric subsection.

The summation of fixed-length information (derived directly from the ID table) plus variable-length information (derived from encoded bits as just described) results in a "known-length entry" for each of the always-numeric strings encoded in the current Packed Object. Each all-numeric data string in a Packed Object (if described as all-numeric in the ID Table) is encoded by converting the digit string into a single Binary number (up to

160 bits, representing a binary value between 0 and (10^{48} – 1)). Figure K.1 — Required number of bits for a given number of Base 'N' values shows the number of bits required to represent a given number of digits. If an all-numeric string contains more than 48 digits, then the first 48 are encoded as one 160-bit group, followed by the next group of up to 48 digits, and so on. Finally, the Binary values for each all-numeric data string in the Object are themselves concatenated to form the Known-length-Numerics subsection.

I.8.2 Alphanumeric subsection of the Data section

The Alphanumeric (A/N) subsection, if present, encodes all of the Packed Object's data from any data strings that were not already encoded in the Known-length Numerics subsection. If there are no alphanumeric characters to encode, the entire A/N subsection is omitted. The Alphanumeric subsection can encode any mix of digits and non-digit ASCII characters, or eight-bit data. The digit characters within this data are encoded separately, at an average efficiency of 4.322 bits per digit or better, depending on the character sequence. The non-digit characters are independently encoded at an average efficiency that varies between 5.91 bits per character or better (all uppercase letters), to a worst-case limit of 9 bits per character (if the character mix requires Base 256 encoding of non-numeric characters).

An Alphanumeric subsection consists of a series of A/N Header bits (see Annex I.8.2.1), followed by from one to four Binary segments (each segment representing data encoded in a single numerical Base, such as Base 10 or Base 30, see Annex I.8.2.4), padded if necessary to complete the final byte (see Annex I.8.2.5).

I.8.2.1 A/N Header Bits

The A/N Header Bits are defined as follows:

- One or two Non-Numeric Base bits, as follows:
  - '0' indicates that Base 30 was chosen for the non-numeric Base;
  - '10' indicates that Base 74 was chosen for the non-numeric Base;
  - '11' indicates that Base 256 was chosen for the non-numeric Base
- Either a single '0' bit (indicating that no Character Map Prefix is encoded), or a '1' bit followed by one or more "Runs" of six Prefix bits as defined in Annex I.8.2.3.
- Either a single '0' bit (indicating that no Character Map Suffix is encoded), or a '1' bit followed by one or more "Runs" of six Suffix bits as defined in Annex I.8.2.3.
- A variable-length "Character Map" bit pattern (see Annex I.8.2.2), representing the base of each of the data characters, if any, that were not accounted for by a Prefix or Suffix.

I.8.2.2 Dual-base Character-map encoding

Compaction of the ordered list of alphanumeric data strings (excluding those data strings already encoded in the Known-Length Numerics subsection) is achieved by first concatenating the data characters into a single data string (the individual string lengths have already been recorded in the Aux Format section). Each of the data characters is classified as either Base 10 (for numeric digits), Base 30 non-numerics (primarily uppercase A-Z), Base 74 non-numerics (which includes both uppercase and lowercase alphas, and other ASCII characters), or Base 256 characters. These character sets are fully defined in Annex K. All characters from the Base 74 set are also accessible from Base 30 via the use of an extra "shift" value (as are most of the lower 128 characters in the Base 256 set). Depending on the relative percentage of "native" Base 30 values vs. other values in the data string, one of those bases is selected as the more efficient choice for a non-numeric base.

Next, the precise sequence of numeric and non-numeric characters is recorded and encoded, using a variable-length bit pattern, called a "character map," where each '0' represents a Base 10 value (encoding a digit) and each '1' represents a value for a non-numeric character (in the selected base). Note that, if (for example) Base 30 encoding was selected, each data character (other than uppercase letters and the space

character) needs to be represented by a pair of base-30 values, and thus each such data character is represented by a *pair* of '1' bits in the character map.

I.8.2.3 Prefix and Suffix Run-Length encoding

For improved efficiency in cases where the concatenated sequence includes runs of six or more values from the same base, provision is made for optional run-length representations of one or more Prefix or Suffix "Runs" (single-base character sequences), which can replace the first and/or last portions of the character map. The encoder shall not create a Run that separates a Shift value from its next (shifted) value, and thus a Run always represents an integral number of source characters.

An optional Prefix Representation, if present, consists of one or more occurrences of a Prefix Run. Each Prefix Run consists of one Run Position bit, followed by two Basis Bits, then followed by three Run Length bits, defined as follows:

— The Run Position bit, if '0', indicates that at least one more Prefix Run is encoded following this one (representing another set of source characters to the right of the current set). The Run Position bit, if '1', indicates that the current Prefix Run is the last (rightmost) Prefix Run of the A/N subsection.

— The first basis bit indicates a choice of numeric vs. non-numeric base, and the second basis bit, if '1', indicates that the chosen base is extended to include characters from the "opposite" base. Thus, '00' indicates a run-length-encoded sequence of base 10 values; '01' indicates a sequence that is primarily (but not entirely) digits, encoded in Base 13; '10' indicates a sequence a sequence of values from the non-numeric base that was selected earlier in the A/N header, and '11' indicates a sequence of values primarily from that non-numeric base, but extended to include digit characters as well. Note an exception: if the non-numeric base that was selected in the A/N header is Base 256, then the "extended" version is defined to be Base 40.

— The 3-bit Run Length value assumes a minimum useable run of six same-base characters, and the length value is further divided by 2. Thus, the possible 3-bit Run Length values of 0, 1, 2, … 7 indicate a Run of 6, 8, 10, … 20 characters from the same base. Note that a trailing "odd" character value at the end of a same-base sequence must be represented by adding a bit to the Character Map.

An optional Suffix Representation, if present, is a series of one or more Suffix Runs, each identical in format to the Prefix Run just described. Consistent with that description, note that the Run Position bit, if '1', indicates that the current Suffix Run is the last (rightmost) Suffix Run of the A/N subsection, and thus any preceding Suffix Runs represented source characters to the left of this final Suffix Run.

I.8.2.4 Encoding into Binary Segments

Immediately after the last bit of the Character Map, up to four binary numbers are encoded, each representing all of the characters that were encoded in a single base system. First, a base-13 bit sequence is encoded (if one or more Prefix or Suffix Runs called for base-13 encoding). If present, this bit sequence directly represents the binary number resulting from encoding the combined sequence of all Prefix and Suffix characters (in that order) classified as Base 13 (ignoring any intervening characters not thus classified) as a single value, or in other words, applying a base 13 to Binary conversion. The number of bits to encode in this sequence is directly determined from the number of base-13 values being represented, as called for by the sum of the Prefix and Suffix Run lengths for base 13 sequences. The number of bits, for a given number of Base 13 values, is determined from the Figure in Annex K. Next, an Extended-NonNumeric Base segment (either Base-40 or Base 84) is similarly encoded (if any Prefix or Suffix Runs called for Extended-NonNumeric encoding).

Next, a Base-10 Binary segment is encoded that directly represents the binary number resulting from encoding the sequence of the digits in the Prefix and/or character map and/or Suffix (ignoring any intervening non-digit characters) as a single value, or in other words, applying a base 10 to Binary conversion. The number of bits to encode in this sequence is directly determined from the number of digits being represented, as shown in Annex K.

Immediately after the last bit of the Base-10 bit sequence (if any), a non-numeric (Base 30, Base 74, or Base 256) bit sequence is encoded (if the character map indicates at least one non-numeric character). This bit sequence represents the binary number resulting from a base-30 to Binary conversion (or a Base-74 to Binary conversion, or a direct transfer of Base-256 values) of the sequence of non-digit characters in the data (ignoring any intervening digits). Again, the number of encoded bits is directly determined from the number of non-numeric values being represented, as shown in Annex K. Note that if Base 256 was selected as the non-Numeric base, then the encoder is free to classify and encode each digit either as Base 10 or as Base 256 (Base 10 will be more efficient, unless outweighed by the ability to take advantage of a long Prefix or Suffix).

Note that an Alphanumeric subsection ends with several variable-length bit fields (the character map, and one or more Binary sections representing the numeric and non-numeric Binary values). Note further that none of the lengths of these three variable-length bit fields are explicitly encoded (although one or two Extended-Base Binary segments may also be present, these have known lengths determined from Prefix and/or Suffix runs). In order to determine the boundaries between these three variable-length fields, the decoder needs to implement a procedure, using knowledge of the remaining number of data bits, in order to correctly parse the Alphanumeric subsection. An example of such a procedure is described in Annex M.

I.8.2.5 Padding the last byte

The last (least-significant) bit of the final Binary segment is also the last significant bit of the Packed Object. If there are any remaining bit positions in the last byte to be filled with pad bits, then the most significant pad bit shall be set to '1', and any remaining less-significant pad bits shall be set to '0'. The decoder can determine the total number of non-pad bits in a Packed Object by examining the Length Section of the Packed Object (and if the Pad Indicator bit of that section is '1', by also examining the last byte of the Packed Object).

I.9 ID Map and Directory encoding options

An ID Map can be more efficient than a list of ID Values, when encoding a relatively large number of ID Values. Additionally, an ID Map representation is advantageous for use in a Directory Packed Object. The ID Map itself (the first major subsection of every ID Map section) is structured identically whether in a Data or Directory IDMPO, but a Directory IDMPO's ID Map section contains additional optional subsections. The structure of an ID Map section, containing one or more ID Maps, is described in Annex I.9.1, explained in terms of its usage in a Data IDMPO; subsequent sections explain the added structural elements in a Directory IDMPO.

I.9.1 ID Map Section structure

An IDMPO represents ID Values using a structure called an ID Map section, containing one or more ID Maps. Each ID Value encoded in a Data IDMPO is represented as a '1' bit within an ID Map bit field, whose fixed length is equal to the number of entries in the corresponding Base Table. Conversely, each '0' in the ID Map Field indicates the absence of the corresponding ID Value. Since the total number of '1' bits within the ID Map Field equals the number of ID Values being represented, no explicit NumberOfIDs field is encoded. In order to implement the range of functionality made possible by this representation, the ID Map Section contains elements other than the ID Map itself. If present, the optional ID Map Section immediately follows the leading pattern indicating an IDMPO (as was described in I.4.2), and contains the following elements in the order listed below:

- An Application Indicator subsection (see I.5.3.1)
- an ID Map bit field (whose length is determined from the ID Size in the Application Indicator)
- a Full/Restricted Use bit (see I.5.3.2)
- (the above sequence forms an ID Map, which may optionally repeat multiple times)
- a Data/Directory indicator bit,
- an optional AuxMap section (never present in a Data IDMPO), and

— Closing Flag(s), consisting of an “Addendum Flag” bit. If ‘1’, then an Addendum subsection is present at the end of the Object Info section (after the Object Length Information).

These elements, shown in Figure I.6 — ID Map section as a maximum structure (every element is present), are described in each of the next subsections.

First ID Map		Optional additional ID Map(s)		Null App Indicator (single zero bit)	Data/ Directory Indicator Bit	(if Directory) Optional AuxMap Section	Closing Flag Bit(s)
App Indicator	ID Map Bit Field (ends with F/R bit)	App Indicator	ID Map Field (ends with F/R bit)				
See Annex I. 5.3.1	See Annex I.9.1. 1 and I.5.3.2	As previous	As previous	See I.5.3.1		*See* Figure I.7 — Optional AuxMap section structure	Addendum Flag Bit

Figure I.6 — ID Map section

When an ID Map section is encoded, it is always followed by an Object Length and Pad Indicator, and optionally followed by an Addendum subsection (all as have been previously defined), and then may be followed by any of the other sections defined for Packed Objects, except that a Directory IDMPO shall not include a Data section.

I.9.1.1 ID Map and ID Map bit field

An ID Map usually consists of an Application Indicator followed by an ID Map bit field, ending with a Full/Restricted Use bit. An ID Map bit field consists of a single “MapPresent” flag bit, then (if MapPresent is ‘1’) a number of bits equal to the length determined from the ID Size pattern within the Application Indicator, plus one (the Full/Restricted Use bit). The ID Map bit field indicates the presence/absence of encoded data items corresponding to entries in a specific registered Primary or Alternate Base Table. The choice of base table is indicated by the encoded combination of DSFID and Application Indicator pattern that precedes the ID Map bit field. The MSB of the ID Map bit field corresponds to ID Value 0 in the base table, the next bit corresponds to ID Value 1, and so on.

In a Data Packed Object’s ID Map bit field, each ‘1’ bit indicates that this Packed Object contains an encoded occurrence of the data item corresponding to an entry in the registered Base Table associated with this ID Map. Note that the valid encoded entry may be found either in the first (“parentless”) Packed Object of the chain (the one containing the ID Map) or in an Addendum IDLPO of that chain. Note further that one or more data entries may be encoded in an IDMPO, but marked “invalid” (by a Delete entry in an Addendum IDLPO).

An ID Map shall not correspond to a Secondary ID Table instead of a Base ID Table. Note that data items encoded in a “parentless” Data IDMPO shall appear in the same relative order in which they are listed in the associated Base Table. However, additional “out of order” data items may be added to an existing data IDMPO by appending an Addendum IDLPO to the Object.

An ID Map cannot indicate a specific number of instances (greater than one) of the same ID Value, and this would seemingly imply that only one data instance using a given ID Value can be encoded in a Data IDMPO. However, the ID Map method needs to support the case where more two or more encoded data items are from the same identifier “class” (and thus share the same ID Value). The following mechanisms address this need:

— Another data item of the same class can be encoded in an Addendum IDLPO of the IDMPO. Multiple occurrences of the same ID Value can appear on an ID List, each associated with different encoded values of the Secondary ID bits.

— A series of two or more encoded instances of the same “class” can be efficiently indicated by a single instance of an ID Value (or equivalently by a single ID Map bit), if the corresponding Base Table entry defines a “Repeat” Bit (see Annex J.2.2).

An ID Map section may contain multiple ID Maps; a null Application Indicator section (with its AppIndicatorPresent bit set to '0') terminates the list of ID Maps.

I.9.1.2 Data/Directory and AuxMap indicator bits

A Data/Directory indicator bit is always encoded immediately following the last ID Map. By definition, a Data IDMPO has its Data/Directory bit set to '0', and a Directory IDMPO has its Data/Directory bit set to '1'. If the Data/Directory bit is set to '1', it is immediately followed by an AuxMap indicator bit which, if '1', indicates that an optional AuxMap section immediately follows.

I.9.1.3 Closing Flags bit(s)

The ID Map section ends with a single Closing Flag. The final bit of the Closing Flags is an Addendum Flag Bit which, if '1', indicates that there is an optional Addendum subsection encoded at the end of the Object Info section of the Packed Object. If present, the Addendum subsection is as described in Annex I.5.6.

I.9.2 Directory Packed Objects

A Directory Packed Object is an IDMPO whose Directory bit is set to '1'. Its only inherent difference from a Data IDMPO is that it does not contain any encoded data items. However, additional mechanisms and usage considerations apply only to a Directory Packed Object, and these are described in the following subsections.

I.9.2.1 ID Maps in a Directory IDMPO

Although the structure of an ID Map is identical whether in a Data or Directory IDMPO, the semantics of the structure are somewhat different. In a Directory Packed Object's ID Map bit field, each '1' bit indicates that a Data Packed Object in the same data carrier memory bank contains a valid data item associated with the corresponding entry in the specified Base Table for this ID Map. Optionally, a Directory Packed Object may further indicate *which* Packed Object contains each data item (see the description of the optional AuxMap section below).

Note that, in contrast to a Data IDMPO, there is no required correlation between the order of bits in a Directory's ID Map and the order in which these data items are subsequently encoded in memory within a sequence of Data Packed Objects.

I.9.2.2 Optional AuxMap Section (Directory IDMPOs only)

An AuxMap Section optionally allows a Directory IDMPO's ID Map to indicate not only presence/absence of all the data items in this memory bank of the tag, but also which Packed Object encodes each data item. If the AuxMap indicator bit is '1', then an AuxMap section shall be encoded immediately after this bit. If encoded, the AuxMap section shall contain one PO Index Field for each of the ID Maps that precede this section. After the last PO Index Field, the AuxMap Section may optionally encode an ObjectOffsets list, where each ObjectOffset generally indicates the number of bytes from the start of the previous Packed Object to the start of the next Packed Object. This AuxMap structure is shown (for an example IDMPO with two ID Maps) in Figure I.7 — Optional AuxMap section structure.

PO Index Field for first ID Map		PO Index Field for second ID Map		Object Offsets Present bit	Optional ObjectOffsets subsection			
POindex Length	POindex Table	POindex Length	POindex Table		Object Offsets Multiplier	Object1 offset (EBV6)	Object2 offset (EBV6)	…	ObjectN offset (EBV6)

Figure I.7 — Optional AuxMap section structure

Each PO Index Field has the following structure and semantics:

- A three-bit POindexLength field, indicating the number of index bits encoded for each entry in the PO Index Table that immediately follows this field (unless the POindex length is ‘000’, which means that no PO Index Table follows).

- A PO Index Table, consisting of an array of bits, one bit (or group of bits, depending on the POIndexLength) for every bit in the corresponding ID Map of this directory packed object.. A PO Index Table entry (i.e., a “PO Index”) indicates (by relative order) which Packed Object contains the data item indicated by the corresponding ‘1’ bit in the ID Map. If an ID Map bit is '0', the corresponding PO Index Table entry is present but its contents are ignored.

- Every Packed Object is assigned an index value in sequence, without regard as to whether it is a “parentless” Packed Object or a “child” of another Packed Object, or whether it is a Data or Directory Packed Object.

- If the PO Index is within the first PO Index Table (for the associated ID Map) of the Directory “chain”, then:

    - a PO Index of zero refers to the first Packed Object in memory,

    - a value of one refers to the next Packed Object in memory, and so on

    - a value of *m,* where *m* is the largest value that can be encoded in the PO Index (given the number of bits per index that was set in the POindexLength), indicates a Packed Object whose relative index (position in memory) is *m or higher*. This definition allows Packed Objects higher than *m* to be indexed in an Addendum Directory Packed Object, as described immediately below. If no Addendum exists, then the precise position is either *m* or some indeterminate position greater than *m.*

- If the PO Index is not within the first PO Index Table of the directory chain for the associated ID Map (i.e., it is in an Addendum IDMPO), then:

    - a PO Index of zero indicates that a prior PO Index Table of the chain provided the index information,

    - a PO Index of *n* (*n* > 0) refers to the *nth* Packed Object above the highest index value available in the immediate parent directory PO; e.g., if the maximum index value in the immediate parent directory PO refers to PO number “3 or greater,” then a PO index of 1 in this addendum refers to PO number 4

    - a PO Index of *m* (as defined above) similarly indicates a Packed Object whose position is the *mth* position, *or higher*, than the limit of the previous table in the chain.

- If the valid instance of an ID Value is in an Addendum Packed Object, an implementation may choose to set a PO Index to point directly to that Addendum, or may instead continue to point to the Packed Object in the chain that originally contained the ID Value.

NOTE The first approach sometimes leads to faster searching; the second sometimes leads to faster directory updates

After the last PO Index Field, the AuxMap section ends with (at minimum) a single “ObjectOffsets Present” bit. A‘0’ value of this bit indicates that no ObjectOffsets subsection is encoded. If instead this bit is a ‘1’, it is immediately followed by an ObjectOffsets subsection, which holds a list of EBV-6 “offsets” (the number of octets between the start of a Packed Object and the start of the next Packed Object). If present, the ObjectOffsets subsection consists of an ObjectOffsetsMultiplier followed by an Object Offsets list, defined as follows:

— An EBV-6 ObjectOffsetsMultiplier, whose value, when multiplied by 6, sets the total number of bits reserved for the entire ObjectOffsets list. The value of this multiplier should be selected to ideally result in sufficient storage to hold the offsets for the maximum number of Packed Objects that can be indexed by this Directory Packed Object's PO Index Table (given the value in the POIndexLength field, and given some estimated average size for those Packed Objects).

— a fixed-sized field containing a list of EBV-6 ObjectOffsets. The size of this field is exactly the number of bits as calculated from the ObjectOffsetsMultiplier. The first ObjectOffset represents the start of the second Packed Object in memory, relative to the first octet of memory (there would be little benefit in reserving extra space to store the offset of the *first* Packed Object). Each succeeding ObjectOffset indicates the start of the next Packed Object (relative to the previous ObjectOffset on the list), and the final ObjectOffset on the list points to the all-zero termination pattern where the *next* Packed Object may be written. An invalid offset of zero (EBV-6 pattern "000000") shall be used to terminate the ObjectOffset list. If the reserved storage space is fully occupied, it need not include this terminating pattern.

— In applications where the average Packed Object Length is difficult to predict, the reserved ObjectOffset storage space may sometimes prove to be insufficient. In this case, an Addendum Packed Object can be appended to the Directory Packed Object. This Addendum Directory Packed Object may contain null subsections for all but its ObjectOffsets subsection. Alternately, if it is anticipated that the capacity of the PO Index Table will also eventually be exceeded, then the Addendum Packed Object may also contain one or more non-null PO Index fields. Note that in a given instance of an AuxMap section, either a PO Index Table or an ObjectOffsets subsection may be the first to exceed its capacity. Therefore, the first position referenced by an ObjectOffsets list in an Addendum Packed Object need not coincide with the first position referenced by the PO Index Table of that same Addendum. Specifically, in an Addendum Packed Object, the first ObjectOffset listed is an offset referenced to the last ObjectOffset on the list of the "parent" Directory Packed Object.

I.9.2.3 Usage as a Presence/Absence Directory

In many applications, an Interrogator may choose to read the entire contents of any data carrier containing one or more "target" data items of interest. In such applications, the positional information of those data items within the memory is not needed during the initial reading operations; only a presence/absence indication is needed at this processing stage. An ID Map can form a particularly efficient Presence/Absence directory for denoting the contents of a data carrier in such applications. A full directory structure encodes the offset or address (memory location) of every data element within the data carrier, which requires the writing of a large number of bits (typically 32 bits or more per data item). Inevitably, such an approach also requires reading a large number of bits over the air, just to determine whether an identifier of interest is present on a particular tag. In contrast, when only presence/absence information is needed, using an ID Map conveys the same information using only one bit per data item defined in the data system. The entire ID Map can be typically represented in 128 bits or less, and stays the same size as more data items are written to the tag.

A "Presence/Absence Directory" Packed Object is defined as a Directory IDMPO that does not contain a PO Index, and therefore provides no encoded information as to where individual data items reside within the data carrier. A Presence/Absence Directory can be converted to an "Indexed Directory" Packed Object (see Annex I.9.2.4) by adding a PO Index in an Addendum Packed Object, as a "child" of the Presence/Absence Packed Object.

I.9.2.4 Usage as an Indexed Directory

In many applications involving large memories, an Interrogator may choose to read a Directory section covering the entire memory's contents, and then issue subsequent Reads to fetch the "target" data items of interest. In such applications, the positional information of those data items within the memory is important, but if many data items are added to a large memory over time, the directory itself can grow to an undesirable size.

An ID Map, used in conjunction with an AuxMap containing a PO Index, can form a particularly efficient "Indexed Directory" for denoting the contents of an RFID tag, and their approximate locations as well. Unlike a

full tag directory structure, which encodes the offset or address (memory location) of every data element within the data carrier, an Indexed Directory encodes a small relative position or index indicating which Packed Object contains each data element. An application designer may choose to also encode the locations of each Packed Object in an optional ObjectOffsets subsection as described above, so that a decoding system, upon reading the Indexed Directory alone, can calculate the start addresses of all Packed Objects in memory.

The utility of an ID Map used in this way is enhanced by the rule of most data systems that a given identifier may only appear once within a single data carrier. This rule, when an Indexed Directory is utilized with Packed Object encoding of the data in subsequent objects, can provide nearly-complete random access to reading data using relatively few directory bits. As an example, an ID Map directory (one bit per defined ID) can be associated with an additional AuxMap “PO Index” array (using, for example, three bits per defined ID). Using this arrangement, an interrogator would read the Directory Packed Object, and examine its ID Map to determine if the desired data item were present on the tag. If so, it would examine the 3 “PO Index” bits corresponding to that data item, to determine which of the first 8 Packed Objects on the tag contain the desired data item. If an optional ObjectOffsets subsection was encoded, then the Interrogator can calculate the starting address of the desired Packed Object directly; otherwise, the interrogator may perform successive read operations in order to fetch the desired Packed Object.

Annex J
(normative)

Packed Objects ID Tables

This Annex defines the Packed Objects ID Table format definition, to be used by all registered ID Tables. These ID Tables, customised for each data system, will be registered with the Registration Authority denoted by ISO/IEC 15961-2.

J.1 Packed Objects Data Format registration file structure

A Packed Objects registered Data Format file consists of a series of "Keyword lines" and one or more ID Tables. Blank lines may occur anywhere within a Data Format File, and are ignored. Also, any line may end with extra blank columns, which are also ignored.

— A Keyword line consists of a Keyword (which always starts with "K-") followed by an equals sign and a character string, which assigns a value to that Keyword. Zero or more space characters may be present on either side of the equals sign. Some Keyword lines shall appear only once, at the top of the registration file, and others may appea multiple times, once for each ID Table in the file.

— An ID Table lists a series of ID Values (as defined in Annex I.5.3). Each row of an ID Table contains a single ID Value (in a required "IDvalue" column), and additional columns may associate Object IDs (OIDs), ID strings, Format strings, and other information with that ID Value. A registration file always includes a single "Primary" Base ID Table, zero or more "Alternate" Base ID Tables, and may also include one or more Secondary ID Tables (that are referenced by one or more Base ID Table entries).

To illustrate the file format, a hypothetical data system registration is shown in Figure J.1 — Hypothetical Data Format registration file. In this hypothetical data system, each ID Value is associated with one or more OIDs and corresponding ID strings. The following subsections explain the syntax shown in the Figure.

J.1.1 File Header section

Keyword lines in the File Header (the first portion of every registration file) may occur in any order, and are as follows:

— **(Mandatory) K-Version = nn.nn**, which the registering body assigns, to ensure that any future revisions to their registration are clearly labelled.

— **(Optional) K-Interpretation = string**, where the "string" argument shall be one of the following: "ISO-646", "UTF-8", "ECI-nnnnnn" (where nnnnnn is a registered six-digit ECI number), ISO-8859-nn, or "UNSPECIFIED". The Default interpretation is "UNSPECIFIED". This keyword line allows non-default interpretations to be placed on the octets of data strings that are decoded from Packed Objects.

— **(Optional) K-ISO15434=nn**, where "nn" represents a Format Indicator (a two-digit numeric identifier) as defined in ISO/IEC 15434. This keyword line allows receiving systems to optionally represent a decoded Packed Object as a fully-compliant ISO/IEC 15434 message. There is no default value for this keyword line.

— **(Optional) K-AppPunc = nn**, where nn represents (in decimal) the octet value of an ASCII character that is commonly used for punctuation in this application. If this keyword line is not present, the default Application Punctuation character is the hyphen.

In addition, comments may be included using the optional Keyword assignment line “K-text = string“, and may appear zero or more times within a File Header or Table Header, but not in an ID Table body.

K-Text = Hypothetical Data Format 100

K-Version = 1.0

K-TableID = F100B0

K-RootOID = urn:oid:1.0.12345.100

K-IDsize = 16

IDvalue	OIDs	IDstring	Explanation	FormatString
0	99	1Z	Legacy ID “1Z” corresponds to OID 99, is assigned IDval 0	14n
1	9%x30-33	7%x42-45	An OID in the range 90..93, Corresponding to ID 7B..7E	1*8an
2	(10)(20)(25)(37)	(A)(B)(C)(D)	a commonly-used set of IDs	(1n)(2n)(3n)(4n)
3	26/27	1A/2B	Either 1A or 2B is encoded, but not both	10n / 20n
4	(30) [31]	(2A) [3B]	2A is always encoded, optionally followed by 3B	(11n) [1*20n]
5	(40/41/42) (53) [55]	(4A/4B/4C) (5D) [5E]	One of A/B/C is encoded, then D, and optionally E	(1n/2n/3n) (4n) [5n]
6	(60/61/(64)[66])	(6A /6B / (6C) [6D])	Selections, one of which includes an Option	(1n / 2n / (3n][4n])

K-TableEnd = F100B0

Figure J.1 — Hypothetical Data Format registration file

J.1.2 Table Header section

One or more Table Header sections (each introducing an ID Table) follow the File Header section. Each Table Header begins with a K-TableID keyword line, followed by a series of additional required and optional Keyword lines, (which may occur in any order) as follows:

— **(Mandatory) K-TableID = FnnXnn**, where **Fnn** represents the ISO-assigned Data Format number (where 'nn' represents one or more decimal digits), and Xnn (where 'X' is either 'B' or 'S') is a registrant-assigned Table ID for each ID Table in the file. The first ID Table shall always be the Primary Base ID Table of the registration, with a Table ID of “B0”. As many as seven additional “Alternate” Base ID Tables may be included, with higher sequential “Bnn” Table IDs. Secondary ID Tables may be included, with sequential Table IDs of the form “Snn”.

— **(Mandatory) K-IDsize = nn:** For a base ID table, the value nn shall be one of the values from the “Maximum number of Table Entries” column of Table I.6. For a secondary ID table, the value nn shall be a power of two (even if not present in Table I.6 — Defined ID Value sizes).

— **(Optional) K-RootOID = urn:oid:i.j.k.ff** where:

  — **I, j, and k** are the leading arcs of the OID (as many arcs as required) and

  — **ff** is the last arc of the Root OID (typically, the registered Data Format number)

  If the K-RootOID keyword is not present, then the default Root OID is **urn:oid:1.0.15961.ff**, where “ff” is the registered Data Format number

— **Other optional Keyword lines:** in order to override the file-level defaults (to set different values for a particular table), a Table Header may invoke one or more of the Optional Keyword lines listed in for the File Header section.

The end of the Table Header section is the first non-blank line that does not begin with a Keyword. This first non-blank line shall list the titles for every column in the ID Table that immediately follows this line; column titles are case-sensitive.

An Alternate Base ID Table, if present, is identical in format to the Primary Base ID Table (but usually represents a smaller choice of identifiers, targeted for a specific application).

A Secondary ID Table can be invoked by a keyword in a Base Table's **OIDs** column. A Secondary ID Table is equivalent to a single Selection list (see Annex J.3) for a single ID Value of a Base ID Table (except that a Secondary table uses K-Idsize to explicitly define the number of Secondary ID bits per ID); the IDvalue column of a Secondary table lists the value of the corresponding Secondary ID bits pattern for each row in the Secondary Table. An **OIDs** entry in a Secondary ID Table shall not itself contain a Selection list nor invoke another Secondary ID Table.

J.1.3 ID Table section

Each ID table consists of a series of one or more rows, each row including a mandatory "IDvalue" column, several defined Optional columns (such as "OIDs", "IDstring", and "FormatString"), and any number of Informative columns (such as the "Explanation" column in the hypothetical example shown above).

Each ID Table ends with a required Keyword line of the form:

K-TableEnd = FnnXnn, where **FnnXnn** shall match the preceding **K-TableID** keyword line that introduced the table.

The syntax and requirements of all Mandatory and Optional columns shall be as described Annex J.2.

J.2 Mandatory and Optional ID Table columns

Each ID Table in a Packed Objects registration shall include an IDvalue column, and may include other columns that are defined in this specification as Optional, and/or Informative columns (whose column heading is not defined in this specification).

J.2.1 IDvalue column (Mandatory)

Each ID Table in a Packed Objects registration shall include an IDvalue column. The ID Values on successive rows shall increase monotonically. However, the table may terminate before reaching the full number of rows indicated by the Keyword line containing **K-IDsize**. In this case, a receiving system will assume that all remaining ID Values are reserved for future assignment (as if the OIDs column contained the keyword "K-RFA"). If a registered Base ID Table does not include the optional OIDs column described below, then the IDvalue shall be used as the last arc of the OID.

J.2.2 OIDs and IDstring columns (Optional)

A Packed Objects registration always assigns a final OID arc to each identifier (either a number assigned in the "OIDs" column as will be described below, or if that column is absent, the IDvalue is assigned as the default final arc). The OIDs column is required rather than optional, if a single IDvalue is intended to represent either a combination of OIDs or a choice between OIDs (one or more Secondary ID bits are invoked by any entry that presents a choice of OIDs).

A Packed Objects registration may include an IDString column, which if present assigns an ASCII-string name for each OID. If no name is provided, systems must refer to the identifier by its OID (see Annex J.4). However, many registrations will be based on data systems that do have an ASCII representation for each defined

Identifier, and receiving systems may optionally output a representation based on those strings. If so, the ID Table may contain a column indicating the IDstring that corresponds to each OID. An empty IDstring cell means that there is no corresponding ASCII string associated with the OID. A non-empty IDstring shall provide a "name" for every OID invoked by the OIDs column of that row (or a single name, if no OIDs column is present). Therefore, the sequence of combination and selection operations in an IDstring shall exactly match those in the row's OIDs column.

A non-empty **OIDs** cell may contain either a keyword, an ASCII string representing (in decimal) a single OID value, or a compound string (in ABNF notation) that a defines a choice and/or a combination of OIDs. The detailed syntax for compound OID strings in this column (which also applies to the IDstring column) is as defined in Annex J.3. Instead of containing a simple or compound OID representation, an OIDs entry may contain one of the following Keywords:

— **K-Verbatim = OIDddBnn**, where "dd" represents the chosen penultimate arc of the OID, and "Bnn" indicates one of the Base 10, Base 40, or Base 74 encoding tables. This entry invokes a number of Secondary ID bits that serve two purposes:

  — They encode an ASCII identifier "name" that might not have existed at the time the table was registered. The name is encoded in the Secondary ID bits section as a series of Base-n values representing the ASCII characters of the name, preceded by a four-bit field indicating the number of Base-n values that follow (zero is permissible, in order to support RFA entries as described below).

  — The cumulative value of these Secondary ID bits, considered as a single unsigned binary integer and converted to decimal, is the final "arc" of the OID for this "verbatim-encoded' identifier.

— **K-Secondary = Snn**, where "Snn" represents the Table ID of a Secondary ID Table in the same registration file. This is equivalent to a Base ID Table row OID entry that contains a single Selection list (with no other components at the top level), but instead of listing these components in the Base ID Table, each component is listed as a separate row in the Secondary ID Table, where each may be assigned a unique OID, IDString, and FormatString.

— **K-Proprietary=OIDdddPnn**, where nn represents a fixed number of Secondary ID bits that encode an optional Enterprise Identifier indicating who wrote the proprietary data (an entry of **K-Proprietary=OIDddP0** indicates an "anonymous" proprietary data item).

— **K-RFA = OIDddBnn**, where "Bnn" is as defined above for Verbatim encoding, except that "B0" is a valid assignment (meaning that no Secondary ID bits are invoked). This keyword represents a Reserved for Future Assignment entry, with an option for Verbatim encoding of the Identifier "name" once a name is assigned by the entity who registered this Data Format. Encoders may use this entry, with a four-bit "verbatim" length of zero, until an Identifier "name" is assigned. A specific FormatString may be assigned to K-RFA entries, or the default a/n encoding may be utilized.

Finally, any OIDs entry may end with a single "**R**" character (preceded by one or more space characters), to indicate that a "Repeat" bit shall be encoded as the last Secondary ID bit invoked by the entry. If '1', this bit indicates that another instance of this class of identifier is also encoded (that is, this bit acts as if a repeat of the ID Value were encoded on an ID list). If '1', then this bit is followed by another series of Secondary ID bits, to represent the particulars of this additional instance of the ID Value.

An IDstring column shall not contain any of the above-listed Keyword entries, and an IDstring entry shall be empty when the corresponding OIDs entry contains a Keyword.

J.2.3 FormatString column (Optional)

An ID Table may optionally define the data characteristics of the data associated with a particular identifier, in order to facilitate data compaction. If present, the FormatString entry specifies whether a data item is all-numeric or alphanumeric (i.e., may contain characters other than the decimal digits), and specifies either a fixed length or a variable length. If no FormatString entry is present, then the default data characteristic is alphanumeric. If no FormatString entry is present, or if the entry does not specify a length, then any length

>=1 is permitted. Unless a single fixed length is specified, the length of each encoded data item is encoded in the Aux Format section of the Packed Object, as specified in Annex I.7.

If a given IDstring entry defines more than a single identifier, then the corresponding FormatString column shall show a format string for each such identifier, using the same sequence of punctuation characters (disregarding concatenation) as was used in the corresponding IDstring.

The format string for a single identifier shall be one of the following:

— A length qualifier followed by “n” (for always-numeric data);

— A length qualifier followed by “an” (for data that may contain non-digits); or

— A fixed-length qualifier, followed by “n”, followed by one or more space characters, followed by a variable-length qualifier, followed by “an”.

A length qualifier shall be either null (that is, no qualifier present, indicating that any length >= 1 is legal), a single decimal number (indicating a fixed length) or a length range of the form “i*j”, where “i” represents the minimum allowed length of the data item, “j” represents the maximum allowed length, and i <= j. In the latter case, if “j” is omitted, it means the maximum length is unlimited.

Data corresponding to an “n” in the FormatString are encoded in the KLN subsection; data corresponding to an “an” in the FormatString are encoded in the A/N subsection.

When a given instance of the data item is encoded in a Packed Object, its length is encoded in the Aux Format section as specified in Annex I.7.2.

The format string for a single identifier shall consist of either the letter “n” (for always-numeric data) or “an” (for data that may contain non-digits), optionally preceded by a length range of the form “i*j”, where “i” represents the minimum allowed length of the data item, and “j” represents the maximum allowed length. When a given instance of the data item is encoded in a Packed Object, its length is encoded in the Aux Format section, using the minimum number of bits that can express the range. The minimum value of the range is not itself encoded, but is specified in the ID Table’s FormatString column.

EXAMPLE

A FormatString entry of “3*6n” indicates an all-numeric data item whose length is always between three and six digits inclusive. A given length is encoded in two bits, where ‘00’ would indicate a string of digits whose length is “3”, and ‘11’ would indicate a string length of six digits.

J.2.4 Interp column (Optional)

Some registrations may wish to specify information needed for output representations of the Packed Object’s contents, other than the default OID representation of the arcs of each encoded identifier. If this information is invariant for a particular table, the registration file may include keyword lines as previously defined. If the interpretation varies from row to row within a table, then an Interp column may be added to the ID Table. This column entry, if present, may contain one or more of the following keyword assignments (separated by semicolons), as were previously defined (see Annex J.1.1 and J.1.2):

— K-RootOID = urn:oid:i.j.k.l…

— K-Interpretation = string

— K-ISO15434=nn

If used, these override (for a particular Identifier) the default file-level values and/or those specified in the Table Header section.

J.3 Syntax of OIDs, IDString, and FormatString columns

In a given ID Table entry, the OIDs, IDString, and FormatString column may indicate one or more mechanisms described in this clause. Annex J.3.1 specifies the semantics of the mechanism and Annex J.3.2 specifies the formal grammar for the ID Table columns.

J.3.1 Semantics for OIDs, IDString, and FormatString columns

In the descriptions below, the word "Identifier" means either an OID final arc (in the context of the OIDs column) or an IDString name (in the context of the IDstring column). If both columns are present, only the OIDs column actually invokes Secondary ID bits.

— A ***Single component*** resolving to a single Identifier, in which case no additional Secondary ID bits are invoked.

— (For OIDs and IDString columns only) A single component resolving to one of a series of closely-related Identifiers, where the Identifier's string representation varies only at one or more character positions. This is indicated using the ***Concatenation*** operator '%' to introduce a range of ASCII characters at a specified position. For example, an OID whose final arc is defined as "391n", where the fourth digit 'n' can be any digit from '0' to '6' (ASCII characters 30_{16} to 36_{16} inclusive) is represented by the component **391%x30-36** (note that no spaces are allowed). A Concatenation invokes the minimum number of Secondary ID digits needed to indicate the specified range. When both an OIDs column and an IDstring column are populated for a given row, both shall contain the same number of concatations, with the same ranges (so that the numbers and values of Secondary ID bits invoked are consistent). However, the minimum value listed for the two ranges can differ, so that (for example) the OID's digit can range from 0 to 3, while the corresponding IDstring character can range from "B" to "E" if so desired. Note that the use of Concatenation inherently constrains the relationship between OID and legacy ID, and so Concatenation may not be useable under all circumstances (the Selection operation described below usually provides an alternative).

— A ***Combination*** of two or more identifier components in an ordered sequence, indicated by surrounding each component of the sequence with parentheses. For example, an IDstring entry **(A)(%x30-37B)(2C)** indicates that the associated ID Value represents a sequence of the following three identifiers:

  — Identifier "A", then

  — An identifier within the range "0B" to "7B" (invoking three Secondary ID bits to represent the choice of leading character), then

  — Identifier "2C"

  Note that a Combination does not itself invoke any Secondary ID bits (unless one or more of its components do).

— An ***Optional*** component is indicated by surrounding the component in brackets, which may viewed as a "conditional combination." For example the entry (A) [B][C][D] indicates that the ID Value represents identifier A, optionally followed by B, C, and/or D. A list of Options invokes one Secondary ID bit for each component in brackets, wherein a '1' indicates that the optional component was encoded.

— A ***Selection*** between several mutually-exclusive components is indicated by separating the components by forward slash characters. For example, the IDstring entry **(A/B/C/(D)(E))** indicates that the fully-qualified ID Value represents a single choice from a list of four choices (the fourth of which is a Combination). A Selection invokes the minimum number of Secondary ID bits needed to indicate a choice from a list of the specified number of components.

In general, a “compound” OIDs or IDstring entry may contain any or all of the above operations. However, to ensure that a single left-to-right parsing of an OIDs entry results in a deterministic set of Secondary ID bits (which are encoded in the same left-to-right order in which they are invoked by the OIDs entry), the following restrictions are applied:

— A given Identifier may only appear once in an OIDs entry. For example, the entry (A)(B/A) is invalid

— A OIDs entry may contain at most a single Selection list

— There is no restriction on the number of Combinations (because they invoke no Secondary ID bits)

— There is no restriction on the total number of Concatenations in an OIDs entry, but no single Component may contain more than two Concatenation operators.

— An Optional component may be a component of a Selection list, but an Optional component may not be a compound component, and therefore shall not include a Selection list nor a Combination nor Concatenation.

— A OIDs or IDstring entry may not include the characters ‘(‘, ‘)’, ‘[‘, ‘]’, ‘%’, ‘-‘, or ‘/’, unless used as an Operator as described above. If one of these characters is part of a defined data system Identifier “name”, then it shall be represented as a single literal Concatenated character.

J.3.2 Formal Grammar for OIDs, IDString, and FormatString Columns

In each ID Table entry, the contents of the OIDs, IDString, and FormatString columns shall conform to the following grammar for `Expr`, unless the column is empty or (in the case of the OIDs column) it contains a keyword as specified in Annex J.2.2. All three columns share the same grammar, except that the syntax for `COMPONENT` is different for each column as specified below. In a given ID Table Entry, the contents of the OIDs, IDString, and FormatString column (except if empty) shall have identical parse trees according to this grammar, except that the `COMPONENT`s may be different. Space characters are permitted (and ignored) anywhere in an `Expr`, except that in the interior of a `COMPONENT` spaces are only permitted where explicitly specified below.

```
Expr ::= SelectionExpr | "(" SelectionExpr ")" | SelectionSubexpr
SelectionExpr ::= SelectionSubexpr ( "/" SelectionSubexpr )+
SelectionSubexpr ::= COMPONENT | ComboExpr
ComboExpr ::= ComboSubexpr+
ComboSubexpr ::= "(" COMPONENT ")" | "[" COMPONENT "]"
```

For the OIDs column, `COMPONENT` shall conform to the following grammar:

```
COMPONENT_OIDs ::= (COMPONENT_OIDs_Char | Concat)+
COMPONENT_OIDs_Char ::= ("0".."9")+
```

For the IDString column, `COMPONENT` shall conform to the following grammar:

```
COMPONENT_IDString ::= UnquotedIDString | QuotedIDString
UnquotedIDString ::= (UnQuotedIDStringChar | Concat)+
UnquotedIDStringChar ::=
   "0".."9" | "A".."Z" | "a".."z" | "_"
QuotedIDString ::= QUOTE QuotedIDStringConstituent+ QUOTE
```

```
QuotedIDStringConstituent ::=
    " " | "!" | "#".."~" | (QUOTE QUOTE)
```

QUOTE refers to ASCII character 34 (decimal), the double quote character.

When the QuotedIDString form for COMPONENT_IDString is used, the beginning and ending QUOTE characters shall *not* be considered part of the IDString. Between the beginning and ending QUOTE, all ASCII characters in the range 32 (decimal) through 126 (decimal), inclusive, are allowed, except that two QUOTE characters in a row shall denote a single double-quote character to be included in the IDString.

In the QuotedIDString form, a % character does not denote the concatenation operator, but instead is just a percent character included literally in the IDString. To use the concatenation operator, the UnquotedIDString form must be used. In that case, a degenerate concatenation operator (where the start character equals the end character) may be used to include a character into the IDString that is not one of the characters listed for UnquotedIDStringChar.

For the FormatString column, COMPONENT shall conform to the following grammar:

```
COMPONENT_FormatString ::= Range? ("an" | "n")
                         | FixedRange "n" " "+ VarRange "an"
Range ::= FixedRange | VarRange
FixedRange ::= Number
VarRange ::= Number "*" Number?
Number ::= ("0".."9")+
```

The syntax for COMPONENT for the OIDs and IDString columns make reference to Concat, whose syntax is specified as follows:

```
Concat ::= "%" "x" HexChar HexChar "-" HexChar HexChar
HexChar ::= ("0".."9" | "A".."F")
```

The hex value following the hyphen shall be greater than or equal to the hex value preceding the hyphen. In the OIDs column, each hex value shall be in the range 30_{16} to 39_{16}, inclusive. In the IDString column, each hex value shall be in the range 20_{16} to $7E_{16}$, inclusive.

J.4 OID input/output representation

The default method for representing the contents of a Packed Object to a receiving system is as a series of name/value pairs, where the name is an OID, and the value is the decoded data string associated with that OID. Unless otherwise specified by a **K-RootOID** keyword line, the default root OID is **urn:oid:1.0.15961.ff,** where **ff** is the Data Format encoded in the DSFID. The final arc of the OID is (by default) the IDvalue, but this is typically overridden by an entry in the OIDs column. Note that an encoded Application Indicator (see Annex I.5.3.1) may change **ff** from the value indicated by the DSFID.

If supported by information in the ID Table's IDstring column, a receiving system shall translate the OID output into various alternative formats, based on the ID String representation of the OIDs. One such format, as described in ISO/IEC 15434, requires as additional information a two-digit Format identifier; a table registration may provide this information using the **K-ISO15434** keyword as described above.

The combination of the K-RootOID keyword and the OIDs column provides the registering entity an ability to assign OIDs to data system identifiers without regard to how they are actually encoded, and therefore the same OID assignment can apply regardless of the access method.

J.4.1 "ID Value OID" output representation

If the receiving system does not have access to the relevant ID Table (possibly because it is newly-registered), the Packed Objects decoder will not have sufficient information to convert the IDvalue (plus Secondary ID bits) to the intended OID. In order to ease the introduction of new or external tables, encoders have an option to follow "restricted use" rules (see. Annex I.5.3.2).

When a receiving system has decoded a Packed Object encoded following "restricted use" rules, but does not have access to the indicated ID Table, it shall construct an "ID Value OID" in the following format:

urn:oid:1.0.15961.300.ff.bb.idval.secbits

where **1.0.15961.300** is a Root OID with a reserved Data Format of "300" that is never encoded in a DSFID, but is used to distinguish an "ID Value OID" from a true OID (as would have been used if the ID Table were available). The reserved value of 300 is followed by the encoded table's Data Format (**ff**) (which may be different from the DSFID's default), the table ID (**bb**) (always '0', unless otherwise indicated via an encoded Application Indicator), the encoded ID value, and the decimal representation of the invoked Secondary ID bits. This process creates a unique OID for each unique fully-qualified ID Value. For example, using the hypothetical ID Table shown in Annex L (but assuming, for illustration purposes, that the table's specified Root OID is **urn:oid:1.0.12345.9**, then an "AMOUNT" ID with a fourth digit of '2' has a true OID of:

urn:oid:1.0.12345.9.3912

and an "ID Value OID" of

urn:oid:1.0.15961.300.9.0.51.2

When a single ID Value represents multiple component identifiers via combinations or optional components, their multiple OIDs and data strings shall be represented separately, each using the same "ID Value OID" (up through and including the Secondary ID bits arc), but adding as a final arc the component number (starting with "1" for the first component decoded under that IDvalue).

If the decoding system encounters a Packed Object that references an ID Table that is unavailable to the decoder, but the encoder chose not to set the "Restricted Use" bit in the Application Indicator, then the decoder shall either discard the Packed Object, or relay the entire Packed Object to the receiving system as a single undecoded binary entity, a sequence of octets of the length specified in the ObjectLength field of the Packed Object. The OID for an undecoded Packed Object shall be **urn:oid:1.0.15961.301.ff.n**, where "301" is a Data Format reserved to indicate an undecoded Packed Object, "ff" shall be the Data Format encoded in the DSFID at the start of memory, and an optional final arc 'n' may be incremented sequentially to distinguish between multiple undecoded Packed Objects in the same data carrier memory.

Annex K
(normative)

Packed Objects Encoding tables

Packed Objects primarily utilise two encoding bases:

— Base 10, which encodes each of the digits ‘0’ through ‘9’ in one Base 10 value

— Base 30, which encodes the capital letters and selectable punctuation in one Base-30 value, and encodes punctuation and control characters from the remainder of the ASCII character set in two base-30 values (using a Shift mechanism) (see Table K.1 — Base 30 Character set).

For situations where a high percentage of the input data’s non-numeric characters would require pairs of base-30 values, two alternative bases, Base 74 and Base 256, are also defined:

— The values in the Base 74 set correspond to the invariant subset of ISO 646 (which includes the GS1 character set), but with the digits eliminated, and with the addition of GS and <space> (GS is supported for uses other than as a data delimiter).

— The values in the Base 256 set may convey octets with no graphical-character interpretation, or “extended ASCII values” as defined in ISO 8859-1 (which is the default character set for this International standard), or UTF-8 (the interpretation may be set in the registered ID Table for an application). The characters ‘0’ through ‘9’ (ASCII values 48 through 57) are supported, and an encoder may therefore encode the digits either by using a prefix or suffix (in Base 256) or by using a character map (in Base 10). Note that in GS1 data, FNC1 is represented by ASCII <GS> (octet value 29_{10}).

Finally, there are situations where compaction efficiency can be enhanced by run-length encoding of base indicators, rather than by character map bits, when a long run of characters can be classified into a single base. To facilitate that classification, additional “extension” bases are added, only for use in Prefix and Suffix Runs.

— In order to support run-length encoding of a primarily-numeric string with a few interspersed letters, a Base 13 is defined (see Table K.2 — Base 13 Character set).

— Two of these extension bases (Base 40 (see Table K.3 — Base 40 Character set) and Base 84 (see Table K.5 — Base 84 Character Set)) are simply defined, in that they extend the corresponding non-numeric bases (Base 30 and Base 74, respectively) to also include the ten decimal digits. The additional entries, for characters ‘0’ through ‘9’, are added as the next ten sequential values (values 30 through 39 for Base 40, and values 74 through 83 for Base 84).

— The “extended” version of Base 256 is defined as Base 40. This allows an encoder the option of encoding a few ASCII control or upper-ASCII characters in Base 256, while using a Prefix and/or Suffix to more efficiently encode the remaining non-numeric characters.

The number of bits required to encode various numbers of Base 10, Base 16, Base 30, Base 40, Base 74, and Base 84 characters are shown in Figure K.1 — Required number of bits for a given number of Base 'N' values. In all cases, a limit is placed on the size of a single input group, selected so as to output a group no larger than 20 octets.

```
/* Base10 encoding accepts up to 48 input values per group: */
static const unsigned char bitsForNumBase10[] = {
/*  0 -  9 */    0,   4,   7,  10,  14,  17,  20,  24,  27,  30,
/* 10 - 19 */   34,  37,  40,  44,  47,  50,  54,  57,  60,  64,
/* 20 - 29 */   67,  70,  74,  77,  80,  84,  87,  90,  94,  97,
/* 30 - 39 */  100, 103, 107, 110, 113, 117, 120, 123, 127, 130,
/* 40 - 48 */  133, 137, 140, 143, 147, 150, 153, 157, 160  };

/* Base13 encoding accepts up to 43 input values per group: */
static const unsigned char bitsForNumBase13[] = {
/*  0 -  9 */    0,   4,   8,  12,  15,  19,  23,  26,  30,  34,
/* 10 - 19 */   38,  41,  45,  49,  52,  56,  60,  63,  67,  71,
/* 20 - 29 */   75,  78,  82,  86,  89,  93,  97, 100, 104, 108,
/* 30 - 39 */  112, 115, 119, 123, 126, 130, 134, 137, 141, 145,
/* 40 - 43 */  149, 152, 156, 160  };

/* Base30 encoding accepts up to 32 input values per group: */
static const unsigned char bitsForNumBase30[] = {
/*  0 -  9 */    0,   5,  10,  15,  20,  25,  30,  35,  40,  45,
/* 10 - 19 */   50,  54,  59,  64,  69,  74,  79,  84,  89,  94,
/* 20 - 29 */   99, 104, 108, 113, 118, 123, 128, 133, 138, 143,
/* 30 - 32 */  148, 153, 158  };

/* Base40 encoding accepts up to 30 input values per group: */
static const unsigned char bitsForNumBase40[] = {
/*  0 -  9 */    0,   6,  11,  16,  22,  27,  32,  38,  43,  48,
/* 10 - 19 */   54,  59,  64,  70,  75,  80,  86,  91,  96, 102,
/* 20 - 29 */  107, 112, 118, 123, 128, 134, 139, 144, 150, 155,
/* 30      */  160  };

/* Base74 encoding accepts up to 25 input values per group: */
static const unsigned char bitsForNumBase74[] = {
/*  0 -  9 */    0,   7,  13,  19,  25,  32,  38,  44,  50,  56,
/* 10 - 19 */   63,  69,  75,  81,  87,  94, 100, 106, 112, 118,
/* 20 - 25 */  125, 131, 137, 143, 150, 156  };

/* Base84 encoding accepts up to 25 input values per group: */
static const unsigned char bitsForNumBase84[] = {
/*  0 -  9 */    0,   7,  13,  20,  26,  32,  39,  45,  52,  58,
/* 10 - 19 */   64,  71,  77,  84,  90,  96, 103, 109, 116, 122,
/* 20 - 25 */  128, 135, 141, 148, 154, 160  };
```

Figure K.1 — Required number of bits for a given number of Base 'N' values

Table K.1 — Base 30 Character set

Val	Basic set		Shift 1 set		Shift 2 set	
	Char	Decimal	Char	Decimal	Char	Decimal
0	A-Punc[1]	N/A	NUL	0	space	32
1	A	65	SOH	1	!	33
2	B	66	STX	2	“	34
3	C	67	ETX	3	#	35
4	D	68	EOT	4	$	36
5	E	69	ENQ	5	%	37
6	F	70	ACK	6	&	38
7	G	71	BEL	7	‘	39
8	H	72	BS	8	(	40
9	I	73	HT	9	)	41
10	J	74	LF	10	*	42
11	K	75	VT	11	+	43
12	L	76	FF	12	,	44
13	M	77	CR	13	-	45
14	N	78	SO	14	.	46
15	O	79	SI	15	/	47
16	P	80	DLE	16	:	58
17	Q	81	ETB	23	;	59
18	R	82	ESC	27	<	60
19	S	83	FS	28	=	61
20	T	84	GS	29	>	62
21	U	85	RS	30	?	63
22	V	86	US	31	@	64
23	W	87	invalid	N/A	\	92
24	X	88	invalid	N/A	^	94
25	Y	89	invalid	N/A	_	95
26	Z	90	[	91	‘	96
27	Shift 1	N/A	]	93	|	124
28	Shift 2	N/A	{	123	~	126
29	P-Punc[2]	N/A	}	125	invalid	N/A

NOTES 1 **Application-Specified Punctuation** character (Value 0 of the Basic set) is defined by default as the ASCII hyphen character (45_{dec}), but may be redefined by a registered Data Format

2 **Programmable Punctuation** character (Value 29 of the Basic set): the first appearance of P-Punc in the alphanumeric data for a packed object, whether that first appearance is compacted into the Base 30 segment or the Base 40 segment, acts as a <Shift 2>, and also “programs” the character to be represented by second and subsequent appearances of P-Punc (in either segment) for the remainder of the alphanumeric data in that packed object. The Base 30 or Base 40 value immediately following that first appearance is interpreted using the Shift 2 column (Punctuation), and assigned to subsequent instances of P-Punc for the packed object.

Table K.2 — Base 13 Character set

Value	Basic set		Shift 1 set		Shift 2 set		Shift 3 set	
	Char	Decimal	Char	Decimal	Char	Decimal	Char	Decimal
0	0	48	A	65	N	78	space	32
1	1	49	B	66	O	79	$	36
2	2	50	C	67	P	80	%	37
3	3	51	D	68	Q	81	&	38
4	4	52	E	69	R	82	*	42
5	5	53	F	70	S	83	+	43
6	6	54	G	71	T	84	,	44
7	7	55	H	72	U	85	-	45
8	8	56	I	73	V	86	.	46
9	9	57	J	74	W	87	/	47
10	Shift1	N/A	K	75	X	88	?	63
11	Shift2	N/A	L	76	Y	89	_	95
12	Shift3	N/A	M	77	Z	90	<GS>	29

Table K.3 — Base 40 Character set

Val	Basic set		Shift 1 set		Shift 2 set	
	Char	Decimal	Char	Decimal	Char	Decimal
0	See Base 30 Table					
…	…					
29	See Base 30 Table					
30	0	48				
31	1	49				
32	2	50				
33	3	51				
34	4	52				
35	5	53				
36	6	54				
37	7	55				
38	8	56				
39	9	57				

Table K.4 — Base 74 Character Set

Val	Char	Decimal	Val	Char	Decimal	Val	Char	Decimal
0	GS	29	**25**	F	70	50	d	100
1	!	33	**26**	G	71	51	e	101
2	"	34	**27**	H	72	52	f	102
3	%	37	**28**	I	73	53	g	103
4	&	38	**29**	J	74	54	h	104
5	'	39	**30**	K	75	55	i	105
6	(	40	**31**	L	76	56	j	106
7	)	41	**32**	M	77	57	k	107
8	*	42	**33**	N	78	58	l	108
9	+	43	**34**	O	79	59	m	109
10	,	44	**35**	P	80	60	n	110
11	-	45	**36**	Q	81	61	o	111
12	.	46	**37**	R	82	62	p	112
13	/	47	**38**	S	83	63	q	113
14	:	58	**39**	T	84	64	r	114
15	;	59	**40**	U	85	65	s	115
16	<	60	**41**	V	86	66	t	116
17	=	61	**42**	W	87	67	u	117
18	>	62	**43**	X	88	68	v	118
19	?	63	**44**	Y	89	69	w	119
20	A	65	**45**	Z	90	70	x	120
21	B	66	**46**	_	95	71	y	121
22	C	67	**47**	a	97	72	z	122
23	D	68	**48**	b	98	73	Space	32
24	E	69	**49**	c	99			

Table K.5 — Base 84 Character Set

Val	Char	Decimal	Val	Char	Decimal	Val	Char	Decimal
0	FNC1	N/A	**25**	F		50	d	
1-73	See Base 74 Table							
74	0	48	**78**	4	52	82	8	56
75	1	49	**79**	5	53	83	9	57
76	2	50	**80**	6	54			
77	3	51	**81**	7	55			

Annex L
(informative)

Encoding example for Packed Objects

In order to illustrate a number of the techniques that can be invoked when encoding a Packed Object, the following sample input data consists of data elements from a hypothetical data system. This data represents:

— An Expiration date (OID 7) of October 31 2006, represented as a six-digit number 061031.

— An Amount Payable (OID 3n) of 1234.56 Euros, represented as a digit string 978123456 (“978” is the ISO Country Code which will indicate that the amount payable is in Euros). As shown in Table L.1 — Hypothetical Base ID Table, for the example in Annex L, this data element is all-numeric, with at least 4 digits and at most 18 digits. In this example, the OID "3n" will be "32", where the "2" in the data element name indicates the decimal point is located two digits from the right.

— A Lot Number (OID 1) of 1A23B456CD.

The application will present the above input to the encoder as a list of OID/Value pairs. The resulting input data, represented below as a single data string (wherein each OID final arc is shown in parentheses) is:

(7)061031(32)978123456(1)1A23B456CD

The example uses a hypothetical ID Table based on GS1 Application Identifiers. In this hypothetical table, each ID Value is a seven-bit index into the Base ID Table; the entries relevant to this example are shown in Table L.1 — Hypothetical Base ID Table, for the example in Annex L.

Table L.1 — Hypothetical Base ID Table, for the example in Annex L

K-Version = 1.0			
K-TableID = F99B0			
K-RootOID = urn:oid:1.0.15961.99			
K-IDsize = 128			
IDvalue	**OIDs**	**Data Title**	**FormatString**
3	1	BATCH/LOT	1*20an
8	7	USE BY OR EXPIRY	6n
51	3%x30-39	AMOUNT	4*18n
125	(7) (1)	EXPIRY + BATCH/LOT	(6n) (1*20an)
K-TableEnd = F99B0			

Encoding is performed in the following steps:

— Three data elements are to be encoded, using the table.

— As shown in the table’s IDstring column, the combination of OID 7 and OID 1 is efficiently supported (because it is commonly seen in applications), and thus the encoder re-orders the input so that 7 and 1 are adjacent and in the order indicated in the OIDs column:

(7)061031(1)1A23B456CD(32)978123456

Now, this OID pair can be assigned a single ID Value of 125 (decimal). The FormatString column for this entry shows that the encoded data will always consist of a fixed-length 6-digit string, followed by a variable-length alphanumeric string.

— Also as shown in Table L.1 — Hypothetical Base ID Table, for the example in Annex L, OID 3n has an ID Value of 51(decimal). The OIDs column for this entry shows that the OID is formed by concatenating “3” with a suffix consisting of a single character in the range 30_{16} to 39_{16} (i.e., a decimal digit). Since that is a range of ten possibilities, a four-bit number will need to be encoded in the Secondary ID section to indicate which suffix character was chosen. The FormatString column for this entry shows that its data is variable-length numeric; the variable length information will require four bits to be encoded in the Aux Format section.

— Since only a small percentage of the 128-entry ID Table is utilised in this Packed Object, the encoder chooses an ID List format, rather than an ID Map format. As this is the default format, no Format Flags section is required.

— This results in the following Object Info section:

  — EBV-6 (ObjectLength): the value is TBD at this stage of the encoding process

  — Pad Indicator bit: TBD at this stage

  — EBV-3 (numberOfIDs) of 001 (meaning two ID Values will follow)

  — An ID List, including:

    — First ID Value: 125 (decimal) in 7 bits, representing OID 7 followed by OID 1

    — Second ID Value: 51(decimal) in 7 bits, representing OID 3n

— A Secondary ID section is encoded as ‘0010’ indicating the trailing ‘2’ of the 3n OID. It so happens this ‘2’ means that two digits follow the implied decimal point, but that information is not needed in order to encode or decode the Packed Object.

— Next, an Aux Format section is encoded. An initial ‘1’ bit is encoded, invoking the Packed-Object compaction method. Of the three OIDs, only OID (3n) requires encoded Aux Format information: a four-bit pattern of ‘0101’ (representing “six” variable-length digits – as “one” is the first allowed choice, a pattern of “0101” denotes “six”)

— Next, the encoder encodes the first data item, for OID 7, which is defined as a fixed-length six-digit data item. The six digits of the source data string are “061031”, which are converted to a sequence of six Base-10 values by subtracting 30_{16} from each character of the string (the resulting values are denoted as values v_5 through v_0 in the formula below). These are then converted to a single Binary value, using the following formula:

$$10^5 * v_5 + 10^4 * v_4 + 10^3 * v_3 + 10^2 * v_2 + 10^1 * v_1 + 10^0 * v_0$$

According to Figure K.1 — Required number of bits for a given number of Base 'N' values, a six-digit number is always encoded into 20 bits (regardless of any leading zero’s in the input), resulting in a Binary string of:

“0000 11101110 01100111”

— The next data item is for OID 1, but since the table indicates that this OID’s data is alphanumeric, encoding into the Packed Object is deferred until after all of the known-length numeric data is encoded.

— Next, the encoder finds that OID 3n, is defined by Table L.1 — Hypothetical Base ID Table, for the example in Annex L as all-numeric, whose length of 9 (in this example) was encoded as (9 – 4 = 5) into four bits within the Aux Format subsection. Thus, a Known-Length-Numeric subsection is encoded for this data item, consisting of a binary value bit-pattern encoding 9 digits. Using Figure K.1 — Required number of bits for a given number of Base 'N' values, the encoder determines that 30 bits need to be encoded in order to represent a 9-digit number as a binary value. In this example, the binary value equivalent of “978123456” is the 30-bit binary sequence:

“111010010011001111101011000000”

— At this point, encoding of the Known-Length Numeric subsection of the Data Section is complete.

Note that, so far, the total number of encoded bits is (3 + 6 + 1 + 7 + 7 + 4 + 5 + 20 + 30) or 83 bits, representing the IDLPO Length Section (assuming that a single EBV-6 vector remains sufficient to encode the Packed Object’s length), two 7-bit ID Values, the Secondary ID and Aux Format sections, and two Known-Length-Numeric compacted binary fields.

At this stage, only one non-numeric data string (for OID 1) remains to be encoded in the Alphanumeric subsection. The 10-character source data string is “1A23B456CD”. This string contains no characters requiring a base-30 Shift out of the basic Base-30 character set, and so Base-30 is selected for the non-numeric base (and so the first bit of the Alphanumeric subsection is set to ‘0’ accordingly). The data string has no substrings with six or more successive characters from the same base, and so the next two bits are set to ‘00’ (indicating that neither a Prefix nor a Suffix is run-length encoded). Thus, a full 10-bit Character Map needs to be encoded next. Its specific bit pattern is ‘0100100011’, indicating the specific sequence of digits and non-digits in the source data string “1A23B456CD”.

Up to this point, the Alphanumeric subsection contains the 13-bit sequence ‘0000100100011’. From Annex K, it can be determined that lengths of the two final bit sequences (encoding the Base-10 and Base-30 components of the source data string) are 20 bits (for the six digits) and 20 bits (for the four uppercase letters using Base 30). The six digits of the source data string “1A23B456CD” are “123456”, which encodes to a 20-bit sequence of:

“00011110001001000000”

which is appended to the end of the 13-bit sequence cited at the start of this paragraph.

The four non-digits of the source data string are “ABCD”, which are converted (using Table K.1 — Base 30 Character set) to a sequence of four Base-30 values 1, 2, 3, and 4 (denoted as values v_3 through v_0 in the formula below. These are then converted to a single Binary value, using the following formula:

$$30^3 * v_3 + 30^2 * v_2 + 30^1 * v_1 + 30^0 * v_0$$

In this example, the formula calculates as (27000 * 1 + 900 * 2 + 30 * 3 + 1 * 4) which is equal to 070DE (hexadecimal) encoded as the 20-bit sequence “00000111000011011110” which is appended to the end of the previous 20-bit sequence. Thus, the AlphaNumeric section contains a total of (13 + 20 + 20) or 53 bits, appended immediately after the previous 83 bits, for a grand total of 136 significant bits in the Packed Object.

The final encoding step is to calculate the full length of the Packed Object (to encode the EBV-6 within the Length Section) and to pad-out the last byte (if necessary). Dividing 136 by eight shows that a total of 17 bytes are required to hold the Packed Object, and that no pad bits are required in the last byte. Thus, the EBV-6 portion of the Length Section is “010001”, where this EBV-6 value indicates 17 bytes in the Object. Following that, the Pad Indicator bit is set to ‘0’ indicating that no padding bits are present in the last data byte.

The complete encoding process may be summarised as follows:

Original input: (7)061031(32)978123456(1)1A23B456CD

Re-ordered as: (7)061031(1)1A23B456CD(32)978123456

FORMAT FLAGS SECTION: (empty)

OBJECT INFO SECTION:
ebvObjectLen: 010001
paddingPresent: 0
ebvNumIDs: 001
IDvals: 1111101 0110011
SECONDARY ID SECTION:
IDbits: 0010
AUX FORMAT SECTION:
auxFormatbits: 1 0101
DATA SECTION:
KLnumeric: 0000 11101110 01100111 111010 01001100 11111010 11000000
ANheader: 0
ANprefix: 0
ANsuffix: 0
ANmap: 01 00100011
ANdigitVal: 0001 11100010 01000000
ANnonDigitsVal: 0000 01110000 11011110
Padding: none

Total Bits in Packed Object: 136; when byte aligned: 136

Output as: 44 7E B3 2A 87 73 3F 49 9F 58 01 23 1E 24 00 70 DE

Table L.1 — Hypothetical Base ID Table, for the example in Annex L shows the relevant subset of a hypothetical ID Table for a hypothetical ISO-registered Data Format 99.

Annex M
(informative)

Decoding Packed Objects

M.1 Overview

The decode process begins by decoding the DSFID. If the leading two bits indicate the Packed Objects Access Method, then the remainder of this Annex applies. From the remainder of the DSFID, determine the Data Format, which shall be applied as the default Data Format for all of the Packed Objects in this memory. From the Data Format, determine the default ID Table that shall be used to process the ID Values in each Packed Object.

Typically, the decoder takes a first pass through the initial ID Values list, as described earlier, in order to complete the list of identifiers. If the decoder finds any identifiers of interest in a Packed Object (or if it has been asked to report back all the data strings from a tag's memory), then it will need to record the implied fixed lengths (from the ID table) and the encoded variable lengths (from the Aux Format subsection), in order to parse the Packed Object's compressed data. The decoder, when recording any variable-length bit patterns, must first convert them to variable string lengths per the table (for example, a three-bit pattern may indicate a variable string length in the range of two to nine).

Starting at the first byte-aligned position after the end of the DSFID, parse the remaining memory contents until the end of encoded data, repeating the remainder of this section until a Terminating Pattern is reached.

Determine from the leading bit pattern (see Annex I.4) which one of the following conditions applies:

a) there are no further Packed Objects in Memory (if the leading 8-bit pattern is all zeroes, this indicates the Terminating Pattern)

b) one or more Padding bytes are present. If padding is present, skip the padding bytes, which are as described in Annex I, and examine the first non-pad byte.

c) a Directory Pointer is encoded. If present, record the offset indicated by the following bytes, and then continue examining from the next byte in memory

d) a Format Flags section is present, in which case process this section according to the format described in Annex I.

e) a default-format Packed Object begins at this location

If the Packed Object has a Format Flags section, then this section may indicate that the Packed Object is of the ID Map format, otherwise it is of the ID List format. According to the indicated format, parse the Object Information section to determine the Object Length and ID information contained in the Packed Object. See Annex I for the details of the two formats. Regardless of the format, this step results in a known Object length (in bits) and an ordered list of the ID Values encoded in the Packed Object. From the governing ID Table, determine the list of characteristics for each ID (such as the presence and number of Secondary ID bits).

Parse the Secondary ID section of the Object, based on the number of Secondary ID bits invoked by each ID Value in sequence. From this information, create a list of the fully-qualified ID Values (FQIDVs) that are encoded in the Packed Object.

Parse the Aux Format section of the Object, based on the number of Aux Format bits invoked by each FQIDV in sequence.

Parse the Data section of the Packed Object:

a) If one or more of the FQIDVs indicate all-numeric data, then the Packed Object's Data section contains a Known-Length Numeric subsection, wherein the digit strings of these all-numeric items have been encoded as a series of binary quantities. Using the known length of each of these all-numeric data items, parse the correct numbers of bits for each data item, and convert each set of bits to a string of decimal digits.

b) If (after parsing the preceding sections) one or more of the FQIDVs indicate alphanumeric data of nonzero length, then the Packed Object's Data section contains an AlphaNumeric subsection, wherein the character strings of these alphanumeric items have been concatenated and encoded into the structure defined in Annex I. Decode this data using the "Decoding Alphanumeric data" procedure outlined below.

For each FQIDV in the decoded sequence:

a) convert the FQIDV to an OID, by appending the OID string defined in the registered format's ID Table to the root OID string defined in that ID Table (or to the default Root OID, if none is defined in the table);

b) complete the OID/Value pair by parsing out the next sequence of decoded characters. The length of this sequence is determined directly from the ID Table (if the FQIDV is specified as fixed length) or from a corresponding entry encoded within the Aux Format section.

M.2 Decoding Alphanumeric data

Within the Alphanumeric subsection of a Packed Object, the total number of data characters is not encoded, nor is the bit length of the character map, nor are the bit lengths of the succeeding Binary sections (representing the numeric and non-numeric Binary values). As a result, the decoder must implement a procedure using knowledge of the remaining number of data bits, in order to correctly parse the AlphaNumeric section. An example of an appropriate procedure is described in this sub-clause.

When decoding the A/N subsection using this procedure, the decoder will first count the number of non-bitmapped values in each base (as indicated by the various Prefix and Suffix Runs), and (from that count) will determine the number of bits required to encoded these numbers of values in these bases. The procedure can then calculate, from the remaining number of bits, the number of explicitly-encoded character map bits. After separately decoding the various binary fields (one field for each base that was used), the decoder "re-interleaves" the decoded ASCII characters in the correct order.

The A/N subsection decoding procedure is as follows:

Determine the total number of non-pad bits in the Packed Object, as described in Annex I.8.2.

- Keep a count of the total number of bits parsed thus far, as each of the subsections prior to the Alphanumeric subsection is processed
- Parse the initial Header bits of the Alphanumeric subsection, up to but not including the Character Map, and add this number to previous value of TotalBitsParsed.
- Initialize a DigitsCount to the total number of base-10 values indicated by the Prefix and Suffix (which may be zero)
- Initialize an ExtDigitsCount to the total number of base-13 values indicated by the Prefix and Suffix (which may be zero)
- Initialize a NonDigitsCount to the total number of base-30, base-74, or base-256 values indicated by the Prefix and Suffix (which may be zero)
- Initialize an ExtNonDigitsCount to the total number of base-40 or base 84 values indicated by the Prefix and Suffix (which may be zero)

- Calculate Extended-base Bit Counts: Using the tables in Annex K, calculate two numbers:
  - ExtDigitBits, the number of bits required to encode the number of base-13 values indicated by ExtDigitsCount, and
  - ExtNonDigitBits, the number of bits required to encode the number of base-40 (or base-84) values indicated by ExtNonDigitsCount
  - Add ExtDigitBits and ExtNonDigitBits to TotalBitsParsed
- Create a PrefixCharacterMap bit string, a sequence of zero or more quad-base character-map pairs, as indicated by the Prefix bits just parsed. Use quad-base bit pairs defined as follows:
  - '00' indicates a base 10 value;
  - '01' indicates a character encoded in Base 13;
  - '10' indicates the non-numeric base that was selected earlier in the A/N header, and
  - '11' indicates the Extended version of the non-numeric base that was selected earlier
- Create a SuffixCharacterMap bit string, a sequence of zero or more quad-base character-map pairs, as indicated by the Suffix bits just parsed.
- Initialize the FinalCharacterMap bit string and the MainCharacterMap bit string to an empty string
- **Calculate running Bit Counts**: Using the tables in Annex B, calculate two numbers:
  - DigitBits, the number of bits required to encode the number of base-10 values currently indicated by DigitsCount, and
  - NonDigitBits, the number of bits required to encode the number of base-30 (or base 74 or base-256) values currently indicated by NonDigitsCount
- set AlnumBits equal to the sum of DigitBits plus NonDigitBits
- if the sum of TotalBitsParsed and AlnumBits equals the total number of non-pad bits in the Packed Object, then no more bits remain to be parsed from the character map, and so the remaining bit patterns, representing Binary values, are ready to be converted back to extended base values and/or base 10/base 30/base 74/base-256 values (skip to the **Final Decoding** steps below). Otherwise, get the next encoded bit from the encoded Character map, convert the bit to a quad-base bit-pair by converting each '0' to '00' and each '1' to '10', append the pair to the end of the MainCharacterMap bit string, and:
  - If the encoded map bit was '0', increment DigitsCount,
  - Else if '1', increment NonDigitsCount
  - Loop back to the **Calculate running Bit Counts** step above and continue
- **Final Decoding steps:** once the encoded Character Map bits have been fully parsed:
  - Fetch the next set of zero or more bits, whose length is indicated by ExtDigitBits. Convert this number of bits from Binary values to a series of base 13 values, and store the resulting array of values as ExtDigitVals.
  - Fetch the next set of zero or more bits, whose length is indicated by ExtNonDigitBits. Convert this number of bits from Binary values to a series of base 40, base 84, or base 30 values (depending on the selection indicated in the A/N Header), and store the resulting array of values as ExtNonDigitVals.
  - Fetch the next set of bits, whose length is indicated by DigitBits. Convert this number of bits from Binary values to a series of base 10 values, and store the resulting array of values as DigitVals.

- Fetch the final set of bits, whose length is indicated by NonDigitBits. Convert this number of bits from Binary values to a series of base 30 or base 74 or base 256 values (depending on the value of the first bits of the Alphanumeric subsection), and store the resulting array of values as NonDigitVals.
- Create the FinalCharacterMap bit string by copying to it, in this order, the previously-created PrefixCharacterMap bit string, then the MainCharacterMap string, and finally append the previously-created SuffixCharacterMap bit string to the end of the FinalCharacterMap string.
- Create an interleaved character string, representing the concatenated data strings from all of the non-numeric data strings of the Packed Object, by parsing through the FinalCharacterMap, and:
  - For each '00' bit-pair encountered in the FinalCharacterMap, copy the next value from DigitVals to InterleavedString (add 48 to each value to convert to ASCII);
  - For each '01' bit-pair encountered in the FinalCharacterMap, fetch the next value from ExtDigitVals, and use Table K-2 to convert that value to ASCII (or, if the value is a Base 13 shift, then increment past the next '01' pair in the FinalCharacterMap, and use that Base 13 shift value plus the next Base 13 value from ExtDigitVals to convert the pair of values to ASCII). Store the result to InterleavedString;
  - For each '10' bit-pair encountered in the FinalCharacterMap, get the next character from NonDigitVals, convert its base value to an ASCII value using Annex K, and store the resulting ASCII value into InterleavedString. Fetch and process an additional Base 30 value for every Base 30 Shift values encountered, to create and store a single ASCII character.
  - For each '11' bit-pair encountered in the FinalCharacterMap, get the next character from ExtNonDigitVals, convert its base value to an ASCII value using Annex K, and store the resulting ASCII value into InterleavedString, processing any Shifts as previously described.

Once the full FinalCharacterMap has been parsed, the InterleavedString is completely populated. Starting from the first AlphaNumeric entry on the ID list, copy characters from the InterleavedString to each such entry, ending each copy operation after the number of characters indicated by the corresponding Aux Format length bits, or at the end of the InterleavedString, whichever comes first.

Annex N
(normative)

Tag Data Profile encoding

This Annex defines the encoding rules for Tag Data Profiles. Because this **Access-Method** uses what is, in effect, an external 'directory' as represented by the Tag Data Profile Table, header information needs to be encoded on the RFID tag so that the relevant table can be called for encoding and decoding purposes.

N.1 Scope

The encoding scheme is designed to support the encoding of any type of application data using existing or newly registered **Data-Formats**, as follows:

- The data format may be any one of the registered data formats
- It may be a new registration, subject to the rules of ISO/IEC 15961-2
- In the case of small applications that do not justify their own data format, then **Data-Format** = 2 shall be used.

Each Tag Data Profile scheme shall be registered with the RA of ISO/IEC 15961-2, with the data profile number being seen as a detailed qualifier to the **Access-Method**.

The **Access-Method** and **Data-Format** shall be encoded in the **DSFID** in the normal manner as defined in 9.2.5 and 9.2.6 (if applicable), and based on the rules of the particular Tag Driver.

N.2 The Registered Table

A Tag Data Profile table, specified in detail in Annex O, shall be registered with the Registration Authority for ISO/IEC 15961-2. If the Tag Data Profile table is for a data element list in a multiple record (see Annex R.5.3), then the assigned table number shall begin with '9'.

The table is designed to be machine readable and is in two parts. The **table header** identifies:

- A version number for the particular table
- Information to enable interpretation of the input string of data
- The **Data-Format**
- The **Tag-Data-Profile** registration number (which also acts as a unique table ID)
- The **Root-OID** for external communication through ISO/IEC 15961 and ISO/IEC 24791 Parts 2 and 5
- The number of **Relative-OIDs** encoded as part of the particular **Tag-Data-Profile** scheme.

The url for the table does not need to be on the table, but does need to be part of the registration process.

The main part of the table has an entry for each data element with the following column headings:

— The sequence number of the data element encoded on the tag, starting from zero

— The **Relative-OID** associated with that data element. By having the two columns, the organisation of data elements can be in pre-determined groups, for example the more important ones in the first positions, or to group locked and unlocked data elements together.

— A lock data element indicator.

— A block align indicator.

— The length and format of the user data.

— The code for the ISO/IEC 15962 basic compaction scheme that is applied to the user data, including UTF-8 and application-defined.

— The compacted length (in bytes).

N.3 Encoding the Tag Data Profile on the RFID tag

The encoding on the tag consists of two segments: a header segment and the data encoding segment. It is more effective to carry out a preliminary encoding of the compacted data, then determine the encoding for the header segment and concatenate these two segments.

N.3.1 Header segment

The header segment shall be encoded on the RFID tag and precede any data encoding, and immediately follow the **DSFID**, if this is encoded as part of the used memory.

NOTE For some tag types, the DSFID is encoded in a separate memory area.

The structure of the header segment shall consist (in sequence) of:

— The registered **Tag-Data-Profile** scheme number, using the same structure as used for length encoding as defined in 9.2.10.

NOTE This means that the number of **Tag-Data-Profile** schemes can be extended to over 16,000 in two bytes, but could even extend beyond this point.

— One byte to indicate the length of the encoded header. This includes any padding bytes to align the header on a boundary relevant to the particular type of tag, for example to a word or block boundary.

— One byte to indicate the number of bytes per block. The term block in this case is defined as the minimum size that can be locked for the particular tag.

NOTE For some tag types this is a different value to the one generally used in the International Standard, which is define as the minimum unit that can be read or written. These different values need to be taken into account during the encoding process.

— The encoded length of the entire **Tag-Data-Profile** using the method defined in 9.2.10.

— The **root-OID**, if the **Data-Format** = 2. This shall be encoded as defined in Annex D.4.2. This field is only encoded if **Data-Format** = 2.

NOTE Because the presence of this component is declared by the **Data-Format** and because the size of the **root-OID** is self-declaring it can be encoded in this sequence in the structure and subsequently decoded.

— Any padding to the end of the header, using byte 80_{16}. This field is only encoded to achieve some alignment (e.g. for achieving a block boundary between the tag header and the encoded data.

This structure for the header information enables the **Tag-Data-Profile** ID Table to be logical in structure and for tags with different block sizes to be used with the same **Tag-Data-Profile**.

N.3.2 Encoded data segment

This particular segment simply consists of a contiguous byte string representing the compacted bytes supported by any necessary pad bytes (byte 80_{16}) for block alignment. There are two main reasons for aligning to a block boundary:

— to ensure that all the bytes of a **Data-Set** that is intended to be locked are properly aligned with adjacent **Data-Sets**, and or

— to align a **Data-Set** to begin on a block boundary for faster read access to that data.

ISO/IEC 15961-1 commands and ISO/IEC 24791-2 messages provide information about object identifiers in their processes. These are used to communicate to the encoding process, which shall follow the rules defined in the table with each data element encoded in the sequence defined by the registered **Tag-Data-Profile** scheme table. If any data element is missing in the initial command or its length is longer, then the entire process shall be considered to be an error.

N.3.2.1 Basic compaction

Each data **Object** shall be compacted according to the declared compaction code. As long as the character set is supported, the compaction shall over-ride any optimisation rules for the **No-Directory Access-Method** for selecting between compaction schemes. For example, an all-numeric string may be defined as requiring integer compaction, even if this begins with a leading zero. This deterministic approach is to ensure a constant size of compacted data for each data **Object**.

As the encoded length is required to declared for the **Tag-Data-Profile**, the following additional rules shall apply:

— If the declared compaction scheme is integer (code 001) leading zeros shall be inserted to make up the length

— If the declared compaction scheme is any other, then the compaction shall satisfy the required length, otherwise there is an error

N.3.2.2 Encoding to the Logical Memory

The first encoded compacted **Data-Set** always begins on a block boundary.

A three-step process is required to establish the starting position of the next and subsequent compacted **Data-Sets**:

1. Taking into account the lock-block size of the tag being encoded, determine whether the previous data **Data-Set** ends on a lock-block boundary.

   — If so mark-up for the previous **Data-Set** all the blocks to be locked or unlocked as specified by the profile. Compact the current **Object** and evaluate as from the beginning of this step

   — Else continue at Step 2

2. Consider the BOOLEAN arguments for locking this **Data-Set** and the previous **Data-Set**.

   — If they are different, invoke the **Offset** (see D.7) and insert, if necessary the required number of pad bytes (80_{16}) after the previous **Object** to reach a lock-block boundary. Compact this **Object** and evaluate as from the beginning of step 1

   — If they are the same, continue at step 3

3. Consider the BOOLEAN argument for write-block aligning this **Data-Set**.

   — If TRUE, invoke the **Offset** and any necessary pad bytes to the previous **Data-Set** to reach a write-block boundary. Compact this **Object** and evaluate as from the beginning of step 1

   — If FALSE, compact this **Object** and evaluate as from the beginning of step 1.

In practice, because the number of lock-block and write-block sizes are likely to be few for a particular tag, a set of basic pro forma structures can be established for encoding to a particular Tag-Data-Profile. Furthermore, for those tag types where the lock-block and the write-block are identical, the process is even more deterministic and simpler to implement.

N.4 Decoding the Tag Data Profile

N.4.1 Decoding Tag Data Profiles with an ID Table

Various strategies can be used to decode the **Tag-Data-Profile**. In a system that only expects the **Tag-Data-Profile**, many of the steps discussed below can be omitted. In a more general situation where the encoding might or might not be based on a **Tag-Data-Profile**, the following process is considered appropriate:

1. If the RFID tag supports the encoding and reading of the **DSFID** from a separate memory, then the Access-Method '11' will clearly indicate encoding compliant with **Tag-Data-Profiles**. If the **DSFID** is not encoded and read as part of a separate process, it has to be captured as a preamble to the header segment.

2. Read a limited amount of the user data to establish the **Access-Method**, **Data-Format** and other information from the header segment. Effectively the generic reading process has to capture the complete header segment before decoding can begin.

   NOTE Unless the **Data-Format** = 2, where the length of the **root-OID** can vary, reading 8 bytes should return the header segment in the majority of cases.

3. Identify the length of encoding, and if this is considered to be capable of being read by the application with one air interface transaction, read the appropriate number of blocks using the most efficient air interface transfer option.

4. If the encoding is large and only selected data is required, use the **Tag-Data-Profile** scheme to identify the block(s) that contain the appropriate data elements and return these. The sequence and number of compacted bytes is specified in the ID Table. To determine the actual location, account has to be taken of the block size and any alignment for locked and block-aligned data. Such calculations can be achieved using information in the ID Table and the block size information and end point of the header segment from the header segment in the tag. This process may be implemented in the interrogator or in application software.

5. Use the EncodeLength data (from the ID Table) to select the relevant compacted bytes.

6. De-compact these using the ISO/IEC 15962 basic de-compaction rule as declared by the CompactCode (from the ID Table).

7. Associate the de-compacted data element to the object identifier and process any response.

N.4.2 Decoding Tag Data Profiles without an ID Table

Because the structure of the encoded header segment is self-declaring, it is possible to read all the encoded blocks on the RFID tag (i.e. those where at least one byte is non-zero) and carry out some basic processing.

If **Data Format** = 2, then the **root-OID** is encoded. The encoding ensures that the structure and length are self-declaring.

Once the details of the header segment have been decoded, the **Data-Sets** are decoded in the same manner as for bytes encoded to a **No-Directory** structure. It is possible to partially decode **Data-Sets** that are not requested by the application command by only processing the Precursor and length of encoded data until the target Relative-OID is found.

N.5 Modifying Data

If the **Tag-Data-Profile** scheme supports the ability for the application to modify data, then this can be achieved, so long as the encoded data remains the same encoded length. Additional pad bytes, as used in the tag header, cannot be used because the decoder is always expecting a fixed number of encoded bytes.

Annex O
(normative)

Tag Data Profile ID tables

This Annex defines the **Tag-Data-Profile** ID Table format definition, to be used by all registered ID Tables. These ID Tables, customised for each data system, will be registered with the Registration Authority denoted by ISO/IEC 15961-2.

O.1 Tag-Data-Profile Data-Format registration file structure

A **Tag-Data-Profile** registered **Data-Format** file consists of a series of “Keyword lines” and one ID Table.

A Keyword line consists of a Keyword (which always starts with “K-“) followed by an equals sign and a character string, which assigns a value to that Keyword. Some Keyword lines shall appear only once, at the op of the registration file, and others may appear multiple times, once for each ID Table in the file.

An ID Table lists a series of ID Values. Each row of an ID Table contains a set of columns that identify the input data and the encoded structure:

- IDValue
- **OID (**defining the **Relative-OID**)
- LockData, a BOOLEAN argument and if TRUE indicating that the data, when encoded, shall be locked
- BlockAlign, a BOOLEAN argument and if TRUE indicating that the encoded data shall begin on a block boundary
- FormatString to define the input data
- CompactCode to define the 15962 basic compaction scheme
- EncodeLength

To illustrate the file format, a hypothetical data system registration is shown in Figure O.1 — Hypothetical Data Format registration file. In this hypothetical data system, each ID Value is associated with one OID and other features. This example includes five OIDs with the following constraints:

- "1" for 18 digit serialised traceability code that shall be locked
- "7" for a 6-character batch number
- "9" for an expiry date 8 digit format YYYYMMDD that shall be write block aligned
- "17" for the manufacturing plant as a 3-alpha code
- "14" for a 5-byte hex code

K-Version = 1.0						
K-Interpretation = ISO-8859-1						
K-ProfileID = F150P123						
K-Root-OID = urn:oid:1.0.12345						
K-Idsize = 5						
IDvalue	**OID**	**LockData**	**BlockAlign**	**FormatString**	**CompactCode**	**EncodeLength**
00	1	T	T	18 DIGIT	001	8
01	7	F	F	6 CHAR	100	5
02	9	F	T	8 DIGIT	001	4
03	17	F	F	3 ALPHA	011	2
04	14	F	F	6 HEX	110	3
K-TableEnd = F150P123						

Figure O.1 — Hypothetical Data Format registration file

The following sub-clauses explain the syntax shown in the figure.

O.2 File Header section

The defined Keyword lines in the File Header (the first portion of every registration file) are:

— **(Mandatory) K-Version = nn.nn**, which the registering body assigns, to ensure that any future revisions to their registration are clearly labelled.

— **K-Interpretation = string**, where the "string" argument shall be one of the following: "ISO-646", "ISO-8859-1, "UTF-8". The default is ISO-8859-1 to align the basic processes of this International Standard.

O.3 Table Header section

One or more Table Header sections (each introducing an ID Table) follow the File Header section. Each Table Header begins with a K-TableID keyword line, followed by a series of additional required and optional Keyword lines, as follows:

— **(Mandatory) K-ProfileID = FnnPnn**, where **Fnn** represents the ISO-assigned Data Format number, and **Pnn** is the ISO assigned Table ID for each ProfileID.

— (Mandatory) K-RootOID = urn:oid:i.j.k.ff where:

— **i, j, and k** are the leading arcs of the OID (as many arcs as required) and **ff** is the last arc of the Root OID (which might be the registered Data Format number)

— **(Mandatory) K-IDsize = nn**, where nn represents the number of OIDs in the encoded Tag Data Profile

The end of the Table Header section is the first non-blank line that does not begin with a Keyword. This first non-blank line shall list the titles for every column in the ID Table that immediately follows this line.

O.4 Table Trailer section

Each ID Table ends with a required Keyword line of the form:

> **K-TableEnd = FnnPnn**, where **FnnPnn** shall match the **K-ProfileID** keyword line that introduced the table.

The syntax and requirements of all Mandatory columns shall be as described O.5.

O.5 Mandatory ID Table columns

Each ID Table in a Tag Data Profile registration shall include a number of columns as defined in the following sub-clauses.

O.5.1 IDvalue column

Each ID Table in a Tag Data Profile registration shall include an IDvalue column. The first row shall define the ID Value zero, and successive rows shall increment monotonically until the size defined by the Keyword line containing **K-IDsize**. This column defines the sequence of encoding on the RFID tag.

O.5.2 OID column

Each ID Table in a Tag Data Profile registration shall include an OID column. The OID value may be any Relative-OID specified for the Root-OID.

O.5.3 LockData column

Each ID Table in a Tag Data Profile registration shall include a LockData column. Using a BOOLEAN argument this declares the application requirements to lock the encoded bytes that are derived from the input data.

O.5.4 BlockAlign column

In addition to any block alignment to achieve the correct locking of data, there may be an additional requirement to begin the encoding of a compacted data **Object** on a block boundary, for example to enable air interface commands to call specifically for a particular encoded **Object**.

O.5.5 FormatString column

Each ID Table in a Tag Data Profile registration shall include a FormatString column to define the data characteristics and fixed length of the data associated with a particular identifier, in order to facilitate data compaction. The input characters shall be compliant with the IETF document *RFC 4234 Augmented BNF for Syntax Specifications: ABNF* Table O.1 — DataFormats and 15962 Compactions shows the input data format and the possible ISO/IEC 15962 compaction schemes. For any given data element, one compaction scheme shall be selected and specified in the ID Table.

Table O.1 — DataFormats and 15962 Compactions

Input Data Format	Char value (HEX)	Description	CompactCode
DIGIT	%x30-39	0-9	001 (integer) for numeric strings ≥ 2 digits and with leading digit ≠ 0 010 (numeric) for numeric strings ≥ 2 digits and with leading digit = 0 110 (octet) for single digit string
HEXDIG	DIGIT / "A" / "B" / "C" / "D" / "E" / "F"		110 (octet)
ALPHA	%x45-5A / %x61–7A	A –Z / a-z	011 (5-bit) for A-Z strings ≥ 3 chars 101 (7-bit) for a-z strings ≥ 8 chars 110 (octet) for other shorter strings
VCHAR	%x21-7E	visible (printing) characters	100 (6-bit) for char 20 to $5F_{16}$ ≥ 6 chars 101 (7-bit) for strings ≥ 8 chars 110 (octet) for other shorter strings
CHAR	%x01-7F	any 7-bit US-ASCII character, excluding NUL	100 (6-bit) for char 20 to $5F_{16}$ ≥ 6 chars 101 (7-bit) for strings ≥ 8 chars 110 (octet) for other shorter strings
OCTET	%x00-FF	8 bits of data	110 (octet) for other shorter strings

O.5.6 CompactCode

Each ID Table in a Tag Data Profile registration shall include a CompactCode column. This defines the 15962 compaction scheme that shall be applied to the data string, rather than leave this as an automatic choice by the encoder. The table shows logical choices that should be followed, but there are process rules that differ from the No-Directory encoding. For example, because the ID Table shows the length of the application data it is possible for the integer compaction to be used with numeric strings beginning with 0.

This code is also relevant to decode the encoded bytes.

O.5.7 EncodeLength

Each ID Table in a Tag Data Profile registration shall include an EncodeLength column. This defines the number of bytes required to encode the application data. It is used to parse the encoded bytes to identify the byte string for de-compaction into the application data.

Annex P
(informative)

Encoding example for Tag Data Profile

In order to illustrate a number of the techniques that can be invoked when encoding a **Tag-Data-Profile** the following example is based on the Profile structure as illustrated in Figure O.1 — Hypothetical Data Format registration file, with the following additional inputs:

- The lock-block size is 8 bytes
- The write-block size is 2 bytes
- The **DSFID** is encoded in user memory
- The five data elements have to following values:
  - Relative-OID 1 = 0123456789121345678
  - Relative-OID 7 = AB12XY
  - Relative-OID 9 = 20091231
  - Relative-OID 17 = JPN
  - Relative-OID 14 = A2B380

P.1 Encoded data segment

P.1.1 Encoding the first Data-Set

The first **Data-Set** requires a Precursor that identifies the compaction scheme used and the **Relative-OID**, followed by the length of the compacted data.

The data associated with Relative-OID 1 compacts as 00 2B DC 54 5E 14 D6 4E. It is 8 bytes long, and it is required to be locked. Adding the Precursor and length byte makes for a total length of 10 bytes, which is not block aligned. The Precursor needs to be set to indicate that it is followed by an Offset byte, so has the bit sequence:

- 1 to indicate that an Offset follows
- 001 to indicate the compaction scheme
- 0001 to indicate the Relative-OID

The Precursor has the hexadecimal value 91, and is followed by the Offset byte with the value 05 indicating five pad bytes after the compacted data to align on the boundary of an 8-byte lock block. So the first Data-Set of 16 bytes has the byte string of:

91 05 08 00 2B DC 54 5E 14 D6 4E 80 80 80 80 80

The **Data-Set** is marked-up as requiring locking.

P.1.2 Encoding the second Data-Set

The second **Data-Set** requires a Precursor that identifies the compaction scheme used and the **Relative-OID**, followed by the length of the compacted data.

The data associated with Relative-OID 7 compacts as 04 2C 72 61 98. It is 5 bytes long and does not require to be locked. Adding the Precursor and length byte makes for a total length of 7 bytes, which is not block aligned. As the next **Data-Set** needs to remain unlocked, no pad pads are required for lock-block purposes. However, the next **Data-Set** needs to begin on a write-lock boundary so some re-alignment is required comprising one byte. The Precursor needs to be set to indicate that it is followed by an Offset byte, so has the bit sequence:

— 1 to indicate that an Offset follows

— 100 to indicate the compaction scheme

— 0111 to indicate the Relative-OID

The Precursor has the hexadecimal value C7, and is followed by the Offset byte with the value 00 indicating that of itself it provides sufficient pad bytes. So the second Data-Set of 8 bytes has the byte string of:

C7 00 05 04 2C 72 61 98

The **Data-Set** is marked-up as not requiring locking.

P.1.3 Encoding the third Data-Set

The third **Data-Set** requires a Precursor that identifies the compaction scheme used and the **Relative-OID**, followed by the length of the compacted data.

The data associated with Relative-OID 9 compacts as 01 32 91 5F. It is 4 bytes long and does not require to be locked, but it does require to be aligned for reading and writing. The previous **Data-Set** required block alignment, so this **Data-Set** is encoded from the beginning of a block boundary. Adding the Precursor and length byte makes for a total length of 6 bytes. As this is block aligned on the 2-byte boundary, no Offset and no pad bytes are required. The Precursor so has the bit sequence:

— 0 to indicate that no Offset follows

— 001 to indicate the compaction scheme

— 1001 to indicate the Relative-OID

The Precursor has the hexadecimal value 19, and is followed by the length byte. So the third Data-Set of 8 bytes has the byte string of:

19 04 01 32 91 5F

The **Data-Set** is marked-up as not requiring locking.

P.1.4 Encoding the fourth Data-Set

The fourth **Data-Set** requires a Precursor that identifies the compaction scheme used. As the **Relative-OID** value is greater than 14, the Precursor cannot directly encode this value and signals that the **Relative-OID** is encoded separately as defined in Annex D.4.4.2 with the value 02_{16}. This is then followed by the length of the compacted data.

The data associated with Relative-OID 17 compacts as 54 1C. This 2-byte string is neither locked nor needs to be block aligned. It can be encoded as output from the encoder. Adding the Precursor, the **Relative-OID** and length byte makes for a total length of 5 bytes. The Precursor needs to be set to indicate that it is followed by a directly encoded **Relative-OID**, so has the bit sequence:

— 0 to indicate that no Offset follows

— 011 to indicate the compaction scheme

— 1111 to indicate that the Relative-OID is separately encoded

The Precursor has the hexadecimal value 7F, and is followed by the byte with the value 02 indicating the **Relative-OID** = 17. So the fourth Data-Set of 5 bytes has the byte string of:

7F 02 02 54 1C

The **Data-Set** is marked-up as not requiring locking.

P.1.5 Encoding the fifth Data-Set

Relative-OID 14 was presented with the six hexadecimal characters 'A2B3C4', so these are to be encoded as the three bytes A2 B3 80.

The fifth **Data-Set** requires a Precursor that identifies the compaction scheme used and the **Relative-OID**, followed by the length of the compacted data. The previous **Data-Set** and this one require no block alignment, so this **Data-Set** is abutted at the end of the previous encoding. Adding the Precursor and length byte makes for a total length of 5 bytes. The Precursor has the bit sequence:

— 0 to indicate that no Offset follows

— 110 to indicate the compaction scheme

— 1110 to indicate the Relative-OID

The Precursor has the hexadecimal value 6E, and is followed by the length byte. So the fifth Data-Set of 5 bytes has the byte string of:

6E 03 A2 B3 80

The **Data-Set** is marked-up as not requiring locking.

P.1.6 The complete data segment

The complete encoding is as follows, with the locked bytes shown in bold and alternate **Data-Sets** shaded for illustration purposes:

91	**01**
08	**00**
2B	**DC**
54	**5E**
14	**D6**
4E	**80**
80	**80**
80	**80**
C7	00
05	04
2C	72
61	98
19	04
01	32
91	5F
7F	02
02	54
1C	6E
03	A2
B3	80

P.2 Encoding the header segment

Account needs to be taken that the DSFID is encoded in user memory, and as this has the Data-Format value 150 two bytes are required for the DSFID. For information and completeness of this example:

- The Access-Method = 11, encoded in the leading bits of the first byte
- As the only extension is for the Data-Format, the next bit is 0
- The last five bits are set = 11111 to indicate an extended Data-Format
- Therefore the first byte = 1101111 = DF_{16}
- The second byte encodes the value of the (Data-Format – 32) = $150_{10} – 32_{10} = 118_{10} = 76_{16}$

These two bytes need to be taken into account for block alignment of the header segment. The following can be determined:

a) The ProfileID = 123, which can be encoded in a single byte = 7B

b) The header size cannot be determined until other component values are established

c) The lock-block size is 8 bytes, encoded as 08_{16}

d) The length of the complete Tag Data Profile is {header segment + data segment} = ?? + 24, and still undetermined, but certain to only require one byte

e) As the size of (a + c + d) is only 3 bytes, it is clear the (b) will only require one byte

f) As the DSFID already takes up two bytes of the 8-byte block, b = 6 and is encoded 06_{16}

g) The value of (d) can be determined as 6 + 24 = 30, which is encoded as $1E_{16}$

h) Two pad bytes are required

The complete encoding of the header segment is: 7B 06 08 1E 80 80

Annex Q
(normative)

Basic encoding rules for Multiple-Records Access-Method

Q.1 Overview

The basic encoding rules defined in this annex support the encoding of homogeneous or heterogeneous **Multiple-Records**. These rules also cover generic aspects for encoding hierarchical records, as defined in Annex R.

The **Multiple-Records Access-Method** comprises the following components:

— A MR-header (see Annex Q.2) shall be encoded as the first entry in the Logical Memory and precede all individual records. The MR-header may be locked or unlocked, or selectively locked.

— A series of records that can support: single records, multiple instances of the same type of record, hierarchical records, and data element lists. Each record comprises a preamble followed by the encoding of the data in the record, which complies with the encoding rules of the nominated Access-Method.

— Optionally, a directory, which should be encoded for larger logical memories.

The MR-header, all the records, data element lists, and entries in the directory shall be block aligned. The MR-header may be locked, or unlocked, or selectively locked.

EXAMPLES

Encoding on an ISO/IEC 18000-6 Type C tag shall begin and end on a word boundary.

Encoding on an ISO/IEC 18000-3 Mode 1 tag shall begin and end on a block boundary.

The MR-header and individual records may have an assigned memory capacity that is greater than required for the initial encoding. This is to enable additional data to be correctly encoded within the boundaries defined for the record. This mechanism is also useful to align records, or the MR-header, to some hardware feature such as a lock block boundary, or some file access mechanism.

Q.2 Encoding the Multiple-Records header

The MR-header for the **Multiple-Records Access-Method** shall begin in the lowest addressable block in memory. The MR-header shall comprise a number of mandatory and conditional components defined in the following sub-clauses. The structure is illustrated in Table Q.1 — The sequential byte structure of the Multiple-Records header.