

# Technical Standard

---

## OSI-Abstract-Data Manipulation API (XOM)



THE *Open* GROUP

[This page intentionally left blank]

# *X/Open CAE Specification*

## **OSI-Abstract-Data Manipulation API (XOM)**

*(formerly entitled “API to OSI Object Management (XOM)”)*

*X/Open Company, Ltd.*



© November 1991, X/Open Company Limited and X.400 API Association

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of X/Open Company Limited and the X.400 API Association.

X/Open CAE Specification

OSI-Abstract-Data Manipulation API (XOM)

ISBN: 1 872630 17 0  
X/Open Document Number: XO/CAE/91/080

Set in Palatino by X/Open Company Ltd., U.K.  
Printed by Maple Press, U.K.  
Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited  
Apex Plaza  
Forbury Road  
Reading  
Berkshire, RG1 1AX  
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

## **OSI-ABSTRACT-DATA MANIPULATION API (XOM)**

<b>Chapter</b>	<b>1</b>	<b>INTRODUCTION</b>
	1.1	PURPOSE
	1.2	MOTIVATION
	1.3	LEVELS
	1.4	C NAMING CONVENTIONS
	1.5	OPTIONS
	1.6	CONFORMANCE
	1.7	ABBREVIATIONS
	1.8	ORGANISATION
<b>Chapter</b>	<b>2</b>	<b>INFORMATION ARCHITECTURE</b>
	2.1	INTRODUCTION
	2.2	OBJECTS
	2.3	OBJECT ATTRIBUTES
	2.4	CLASSES
	2.5	CLASS DEFINITIONS
	2.6	PACKAGES
	2.7	PACKAGE DEFINITIONS
	2.8	WORKSPACES
	2.9	STORAGE MANAGEMENT
<b>Chapter</b>	<b>3</b>	<b>INFORMATION SYNTAXES</b>
	3.1	INTRODUCTION
	3.2	SYNTAX TEMPLATES
	3.3	SYNTAXES
	3.4	STRINGS
	3.5	REPRESENTATION OF STRING VALUES
	3.6	RELATIONSHIP TO ASN.1 SIMPLE TYPES
	3.7	RELATIONSHIP TO ASN.1 USEFUL TYPES

	3.8	RELATIONSHIP TO ASN.1 CHARACTER STRING TYPES
	3.9	RELATIONSHIP TO ASN.1 TYPE CONSTRUCTORS
<b>Chapter</b>	<b>4</b>	<b>SERVICE INTERFACE</b>
	4.1	INTRODUCTION
	4.2	DATA TYPES
	4.2.1	Boolean
	4.2.2	Descriptor
	4.2.3	Enumeration
	4.2.4	Exclusions
	4.2.5	Integer
	4.2.6	Modification
	4.2.7	Object
	4.2.8	Object Identifier
	4.2.9	Private Object
	4.2.10	Public Object
	4.2.11	Return Code
	4.2.12	String
	4.2.13	Syntax
	4.2.14	Type
	4.2.15	Type List
	4.2.16	Value
	4.2.17	Value Length
	4.2.18	Value Position
	4.2.19	Workspace
	4.3	FUNCTIONS
	4.4	RETURN CODES
	4.5	DECLARATION SUMMARY
<b>Chapter</b>	<b>5</b>	<b>WORKSPACE INTERFACE</b>
	5.1	INTRODUCTION
	5.2	REPRESENTATION OF OBJECTS
	5.3	TYPES AND MACROS
	5.3.1	Standard Internal Representation of an Object
	5.3.2	Standard Internal Representation of a Workspace
	5.3.3	Useful Macros
	5.4	DISPATCHER MACROS
<b>Chapter</b>	<b>6</b>	<b>OBJECT MANAGEMENT PACKAGE</b>
	6.1	INTRODUCTION
	6.2	CLASS HIERARCHY
	6.3	CLASS DEFINITIONS

## *Contents*

- 6.3.1 Encoding
- 6.3.2 External
- 6.3.3 Object

## **Appendix A GLOSSARY**





# *Preface*

## **X/Open**

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

## **X/Open Technical Publications**

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

Developers who base their products on a current CAE specification can be sure that either the current specification or an upwards-compatible version of it will be referenced by a future X/Open brand (if not referenced already), and that a variety of compatible, X/Open-branded systems capable of hosting their products will be available, either immediately or in the near future.

CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be

developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

### Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.
- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

### Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to [info-server@xopen.co.uk](mailto:info-server@xopen.co.uk) with the following in the Subject line:

request corrigenda; topic index

This will return the index of publications for which Corrigenda exist.

### This Document

This document is a CAE Specification (see above). The previous **Preliminary Specification** version was published with the title *API to OSI Object Management (XOM)*. Note that the title has since been changed, and for this CAE Specification version it is now *OSI-Abstract-Data Manipulation (XOM) API*. The scope and purpose of the Specification, and the *XOM* acronym, remain unchanged.

The *XOM* Specification defines the application programming interface to OSI object management. This interface is intended for use in conjunction with, but is otherwise independent of, other application-specific APIs for Open Systems Interconnection (OSI). In particular, this API is used to manipulate complex arguments and parameters used by the Directory Services (XDS) and Electronic Mail (X.400) APIs.

The content of this specification has been developed in collaboration with the X.400 API Association, which is publishing its own version of the specification.

XOM is one of several specifications that X/Open has developed in collaboration with the X.400 API Association. The other documents are:

- API to Directory Services (XDS)
- API to Electronic Mail (X.400)
- Message Store API
- EDI Messaging Package
- Guide to Selected X.400 and Directory Services APIs.

# *Trademarks*

X/Open and the 'X' device are trademarks of X/Open Company Limited in the U.K. and other countries.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

Palatino is a trademark of Linotype AG and/or its subsidiaries.

## *Referenced Documents*

The following documents define ASN.1 and the BER. Developed under the auspices of, and ratified by both the CCITT and ISO, they provide data type definitions for the interface:

- Recommendation X.208, Specification of Abstract Syntax Notation One (ASN.1), CCITT Blue Book, Fascicle VIII.4, International Telecommunications Union, 1988. (Also published by ISO as ISO 8824.)
- Recommendation X.209, Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), Ibid. (Also published by ISO as ISO 8825.)

*Note: ASN.1 and the BER slightly alter and extend the technology of X.409 (1984). In general, X.208 and X.209, not X.409, underlie the present edition of this document.*

The following document defines, in clause 8.7, the canonical form of the BER. It provides a partial basis for the definition of the Encode and Decode functions:

- Recommendation X.509, The Directory: Authentication Framework, CCITT Blue Book, International Telecommunications Union, 1988. (Also published by ISO as ISO 9594-8.)

The following document defines the C language, and provides the context in which the C interface is defined:

- Draft International Standard ISO/IEC DIS 9899, Programming Languages: C, ISO.

*Note: Programming Language C, X3.159-1989, published by ANSI, forms the basis of this Draft International Standard.*

# Introduction

*Note that the previous **Preliminary Specification** version of the XOM document was published with the title **API to OSI Object Management (XOM)**. The title has since been changed, and for this CAE Specification it is now **OSI-Abstract-Data Manipulation (XOM) API**. The scope and purpose, and the XOM acronym, remain unchanged.*

This Chapter introduces the interface and its specification. It indicates the purpose of the interface, provides the motivation for it, identifies the levels of abstraction at which the interface is defined, explains how identifiers at one level are derived from those at the other, summarises the service implementation options, gives the conformance requirements imposed upon manufacturers and their products, lists the abbreviations used in the document, and describes the document's organisation.

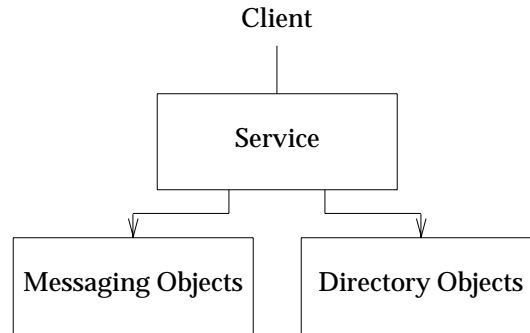
## 1.1 PURPOSE

This document defines a general-purpose OSI Object Management Application Program Interface (API) for use in conjunction with, but otherwise independent of, other application-specific APIs for Open Systems Interconnection (OSI).

Object Management (OM) is the creation, examination, modification and deletion of potentially complex information objects. It presents to programmers a uniform model, or architecture, of information based upon the concept of groups, or classes, of similar information objects. Programmers manipulate objects too large for main memory, for example, in the same way as they manipulate smaller objects.

The information objects to which OM applies are those that arise in OSI, i.e. those that correspond to the types defined by, or by means of, Abstract Syntax Notation One (ASN.1). The OM API comprises tools for manipulating ASN.1 objects. It shields the programmer from much of ASN.1's complexity, e.g. its Basic Encoding Rules (BER).

The OM API is designed to be implemented by one or more manufacturers working independently. As illustrated in Figure 1 below, each manufacturer effectively provides the programmer with the ability to manipulate information objects of particular kinds. This division of implementation responsibility is achieved by means of workspaces (see **Section 2.8, Workspaces**).



**Figure 1: Conceptual Model of Object Management**

Throughout this document, the term ‘interface’ denotes the OM API, the term ‘service’ denotes software that supplies (i.e., implements) the interface, and the term ‘client’ denotes software that uses the interface. The term ‘service interface’ denotes the interface realised by the service as a whole, and is thus a synonym for ‘interface’.

## 1.2 MOTIVATION

The OM interface is designed to be used with other, application-specific APIs, e.g. for message transfer or directory access. Such an API typically defines both a set of functions and a set of structured information objects that serve as the functions’ arguments and results. The OM interface defines a general information architecture for structuring such information objects, as well as general functions for manipulating them. Both the architecture and the functions are independent of the application-specific APIs they support.

The OM information architecture is object-oriented and thus enjoys the modularity and extensibility inherent in that approach. Collections of data are referred to as objects, objects of like structure are grouped together into classes, and classes are related to one another by means of subclassing. The information architecture is described in detail in **Chapter 2, Information Architecture**. Among its important features, however, is that wherever an application-specific API requires an instance of a particular class, an instance of any of that class’ subclasses may be supplied instead. This permits the refinement and extension of objects and the use of the extended objects in application-specific APIs.

*Note that while OM takes an object-oriented view of structured information, it does not incorporate all the characteristics of other object-oriented systems. In particular, the implementations of the functions for manipulating objects are separate from the definitions of the objects’ classes, and there is no notion of encapsulating or hiding the information associated with objects (although the interface hides the information’s representation).*

The OM interface is specifically designed for use with application-specific APIs that provide OSI services. The objects addressed by the information architecture, therefore, are those arising from ASN.1, the descriptive tool used pervasively in OSI. By providing tools for manipulating ASN.1 objects, the OM interface shields the client from much (but not all) of ASN.1’s complexity.

While it presents to the client a single model of information, the OM interface neither defines nor unnecessarily constrains the representation of the objects that the service



maintains internally. The interface is designed as a general information management facility that can accommodate the varied objects required by application-specific OSI APIs. Therefore, no constraints are placed on the structure, size or location of the information objects held by the service. The internal representation of such objects, which may be application-specific, is hidden from the client; the objects can be accessed only using OM interface functions.

Objects are conveyed between the client and the service, in whole or in part, using sequences of descriptors. Unlike that of the objects themselves, the representation of such sequences is part of the OM interface specification.

The extent to which the OM interface is able to hide the internal representation of objects is insufficient to fully meet the needs of environments supporting several application-specific APIs. Such APIs may impose varied and even conflicting requirements upon the internal representations of objects, and may even be implemented by different vendors. Therefore, the OM interface is designed to permit any number of OM interface implementations to coexist, each representing objects differently. This is accomplished by means of workspaces (see **Section 2.8, Workspaces**).

The various OM interface implementations cannot be completely independent. Two different application-specific services using two different OM interface implementations may have to exchange information. A message transfer system, for example, may require a name or address obtained from a directory system. The OM interface enables such information exchange by providing a single OM interface (used by all application-specific services) to any number of implementations of that interface (workspaces). While it recognises that different workspaces handle objects of different kinds, the client need not explicitly move information from workspace to workspace in order to effectively convey it from one application-specific service to another.

### 1.3 LEVELS

This document defines the interface at two levels of abstraction. It defines a generic interface independent of any particular programming language, and a C interface based on the variant of C standardised by the American National Standards Institute (ANSI). (It does not define interfaces specific to other languages.)

The C interface definition provides language-specific declarations beyond the scope of the generic interface definition. For readability alone, the specifications of the generic and C interfaces are physically combined, rather than physically separated.

### 1.4 C NAMING CONVENTIONS

How the identifier for an element of the C interface is derived from the name of the corresponding element of the generic interface depends on the element's type, as specified in Table 1 below. The generic name is prefixed with the character string in the second column of the table, alphabetic characters are converted to the case in the third column, and an underscore ( `_` ) is substituted for each hyphen ( `-` ) or space (  ).

Element Type	Prefix	Case
Data type	OM_	Lower
Data value	OM_	Upper
Data value (Class1)	OM_C_	Upper
Data value (Syntax)	OM_S_	Upper
Data value component (Structure member)	<i>none</i>	Lower
Function	om_	Lower
Function argument	<i>none</i>	Lower
Function result	<i>none</i>	Lower
Macro	OM_	Upper
Reserved for use by implementors	OMP	any
Reserved for use by implementors	omP	any
Reserved for proprietary extension	omX	any
Reserved for proprietary extension	OMX	any

**Table 1: Derivation of C Identifiers**

The prefixes “omP” and “OMP” are reserved for use by implementors of the service. The prefixes “omX” and “OMX” are reserved for the proprietary extension of the interface. In all other respects, such extension is outside the scope of this document.

## 1.5 OPTIONS

The following aspects of the service’s behaviour are implementation-defined:

1. The local character set representation and the precise mappings between it and the various string syntaxes.
2. The precise definitions in C of the intermediate data types.
3. The length of the longest string that the Get function will return. This number is no less than 1024.
4. Whether the service reports an exception if an object supplied to it as an argument is not minimally consistent.

## 1.6 CONFORMANCE

A service manufacturer shall claim conformance to this document only if it and its product collectively satisfy the following requirements:

1. **Version.** The manufacturer shall claim to support the version of the interface defined by the present edition of this document.

*Note that proprietary versions of the interface are outside the document’s scope. However, to denote additional types, syntaxes and exceptions, such versions should employ integers outside of the intervals used by editions of this document.*

2. **Workspaces.** The service shall comprise one or more workspaces (see **Chapter 5, Information Architecture**), each of which shall support the OM package (see **Chapter 6, Object Management Package**).
3. **Aspects.** The service shall implement all defined aspects of the interface, subject to the options summarised in **Section 1.5, Options**.

4. **Encoding Rules.** In its implementation of the Encode and Decode functions, the service shall support the BER.

## 1.7 ABBREVIATIONS

The following abbreviations are used throughout this document.

ANSI	American National Standards Institute
API	Application Program Interface
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CCITT	International Telegraph and Telephone Consultative Committee
IA5	International Alphabet No. 5
ISO	International Organisation for Standardisation
OM	Object Management
OSI	Open Systems Interconnection
UTC	Universal Coordinated Time

## 1.8 ORGANISATION

Subsequent chapters of this document are as follows:

- **Chapter 2, Information Architecture,**
- **Chapter 3, Information Syntaxes,**
- **Chapter 4, Service Interface,**
- **Chapter 5, Information Architecture,**
- **Chapter 6, Object Management Package, and**
- **Appendix A, Glossary.**



# Information Architecture

## 2.1 INTRODUCTION

This Chapter specifies the architecture of the information that the client and service exchange, and that the service maintains and makes accessible to the client.

The architecture provides a basis for specifying, in **Chapter 4, Service Interface** and **Chapter 5, Information Architecture**, how the client communicates with the service, how the service communicates with the client in response, and how components of the service communicate with one another. The architecture's purpose is not to dictate the physical structure of information as the service maintains it internally; this is implementation-specific and thus unspecified.

## 2.2 OBJECTS

The principal purpose of the service is to create, examine, modify and destroy complex information objects under the client's direction. A principal purpose of the interface is to enable the client and service to exchange objects in whole or in part. This requirement provides the rationale for the information architecture whose specification follows.

Objects are of two kinds: public and private. A 'public object' is represented by a data structure whose format is part of the service's specification (see **Section 4.2, Data Types**). A 'private object', on the other hand, is represented in a fashion that is implementation-specific and thus unspecified. The client therefore accesses private objects only indirectly, i.e. by means of interface functions.

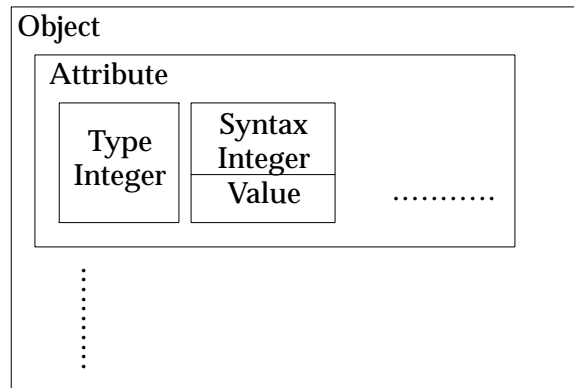
The interface comprises functions for both examining and modifying private objects. For application-specific reasons, however, the service may deny a client request to modify a particular object at a particular time. The specification of each application-specific API shall identify any circumstances under which this may occur.

Public objects themselves are of two kinds, "client-generated" and "service-generated". A client-generated public object is fabricated by the client in storage it provides. A service-generated public object is fabricated by the service in storage it provides. The client creates, examines, modifies, and destroys client-generated public objects directly, i.e. by means of programming language constructs.

*Note that client-generated public objects simplify application programs, enabling them, where appropriate, to statically define objects, rather than requiring them to dynamically construct the objects by means of sequences of interface function calls.*

### 2.3 OBJECT ATTRIBUTES

Objects have internal structure, as illustrated in Figure 2 below. An object comprises zero or more information items called attributes. An 'attribute', in turn, comprises an integer denoting the attribute's type and one or more information items called values, each accompanied by an integer denoting that value's syntax. A 'value' (e.g. one) is an information item, possibly complex, which can be viewed as a characteristic or property of the object of which it is a part. A 'syntax' (e.g. Integer) is a category into which a value is placed on the basis of its form. A 'type' (e.g. Priority) is a category into which all of the values of an attribute are placed on the basis of their purpose. The attribute type is used as the name of the attribute.



**Figure 2: Structure of an Object**

The client and service exchange values by means of descriptors. A 'descriptor' normally comprises a value and the integers that denote the value's syntax and type; sometimes the value is absent (see the Get function).

While syntaxes and types are denoted by integers, the scope of the integers differs. Syntaxes are defined and assigned integers by this document. The integers' scope is global. Types are defined and assigned integers by OM applications. The integers' scope is a package (see **Section 2.6, Packages**).

An object's attributes are unordered, but an attribute's values are ordered. The position of the first value is zero. The positions of successive values are successive positive integers.

One object,  $O_2$ , may be a value of an attribute of another object,  $O_1$ .  $O_2$  is called an 'immediate subobject' of  $O_1$ , and  $O_1$  the 'immediate superobject' of  $O_2$ . The immediate subobjects of  $O_1$ , and all of their subobjects, are the 'subobjects' of  $O_1$ . The immediate superobject of  $O_2$ , and all of its superobjects, are the 'superobjects' of  $O_2$ . The package (see **Section 2.6, Packages**) that contains an object's class may differ from those containing the classes of its immediate subobjects, which may differ from one another.

## 2.4 CLASSES

Objects are categorised on the basis of their purpose and internal structure. Each category is called a 'class'. An object (e.g. a message) is said to be an 'instance' of its class (e.g. Message). A class is characterised by the types of the attributes that may appear in its instances. A class is denoted by an ASN.1 object identifier. The object identifier that denotes a class is an attribute of every instance of the class. In particular, it is the value of the Class attribute, which is specific to the Object class.

*Note that an object identifier may (but need not) be assigned to a class in two steps. First, a distinct integer is assigned to each class in a package (see **Section 2.6, Packages**). Second, the integer is appended to the object identifier assigned to the package, becoming its final subidentifier.*

The types that may appear in an instance of one class,  $C_2$ , are often a superset of those that may appear in an instance of another class,  $C_1$ . When this is so,  $C_2$  may (but need not) be designated a 'subclass' of  $C_1$ , making  $C_1$  a 'superclass' of  $C_2$ . If  $C_1$  is a superclass of no other superclass of  $C_2$ ,  $C_1$  is called the 'immediate superclass' of  $C_2$ , and  $C_2$  an 'immediate subclass' of  $C_1$ . Every class (except Object) is the immediate subclass of exactly one other class; thus the class hierarchy is a tree.

The package (see **Section 2.6, Packages**) containing an object's class may differ from those containing its immediate subclasses, which may differ from one another. The specification of such a class must ensure that each attribute type in the package-closure is allocated a unique integer representation. Specifications produced by X/Open and the X.400 API Association achieve this by use of disjoint sets of integers for each package in all specifications.

The classes form a hierarchy by virtue of the superclass relationships between them. The hierarchy's root is a special class, Object, of which all other classes are subclasses. (Class Object is defined in **Section 6.3.3., Object**). The class hierarchy is fixed by the classes' definitions (see **Section 2.5, Class Definitions**); it cannot be altered programatically.

The types that may appear in an instance of a class but not in an instance of its immediate superclass are said to be 'specific' to the class. Thus the types that may appear in an object are those specific to its class and those specific to each of its superclasses. The set of types that may appear in an object is fixed by the definitions of the classes involved (see **Section 2.5, Class Definitions**); it cannot be altered programatically. The fact that an attribute may appear in instances of a class does not (itself) imply that it must appear (i.e. have a value) in every instance of the class.

An instance of a class is also considered an instance of each of its superclasses, and thus may appear wherever the interface requires an instance of any of those classes.

*Note that this is one of the most useful consequences of the subclassing mechanism.*

Classes are of two kinds, concrete and abstract. Instances of a concrete class are permitted, but instances of an abstract class are forbidden. An abstract class may be defined as a superclass in order to share attributes between classes, or simply to ensure that the class hierarchy is convenient for the interface definition.

The definition of each concrete class may also indicate that the client may not create instances; in this case, instances can only be created as a result of an application-specific function. It is an error for a client to attempt to create an object of such a class (*function-declined*), or of an abstract class (*not-concrete*).

*Note that the OM information architecture has some, but not all, of the important characteristics of object-oriented programming systems. The functions by means of which objects are manipulated, for example, may vary from workspace to workspace (see Section 2.8, Workspaces) but not from class to class.*

## 2.5 CLASS DEFINITIONS

For purposes of the generic interface, and within the context provided by a package (see Section 2.6, Packages), the definition of a class has the following elements:

1. the class' name, which denotes the class' object identifier;
2. identification of the class' immediate superclass;
3. the definitions of the attribute types specific to the class, and
4. an indication of whether the class is abstract or concrete.

The types specific to a class are themselves defined by means of a table (e.g. Table 11 in Chapter 6, Object Management Package). The table gives

- a. under the heading **Attribute**, the name of each attribute
- b. under the heading **Value Syntax**, the syntax or syntaxes of each of its values
- c. under the heading **Value Length**, any constraints upon the number of bits, octets or characters in each value that is a string
- d. under the heading **Value Number**, any constraints upon the number of values
- e. under the heading **Value Initially**, any value the Create function supplies upon request.

With respect to the syntax of attribute values, the designation 'any' denotes any defined syntax.

A class table imposes certain constraints upon instances of the class. The definition of a class may impose additional constraints which may be arbitrarily complex. Such constraints are specified in prose.

Note that classes are often constrained in the following additional ways. Each instance of the class may be constrained to contain exactly one member of a set of attributes. An attribute may be constrained to have

- a. no more than a fixed number of values
- b. either zero or one value, that value thus being optional
- c. exactly one value, that value thus being mandatory.

An attribute's values may be constrained to a single syntax. A syntax may be constrained to a proper subset of its defined values.



An object is said to be 'minimally consistent' if, and only if

- a. the type of each of its attributes is specific to the object's class or one of its superclasses
- b. the number of values of each attribute is no greater than the class permits
- c. the syntax of each value is among those the class permits
- d. the number of bits, octets or characters in each value that is a string is among those the class permits.

Each object that an interface function returns as a result is minimally consistent. Furthermore, the intent of the interface definition is that each object supplied as a function argument is minimally consistent. However, whether the service reports an exception if this is not so is implementation-defined.

## 2.6 PACKAGES

Related classes are grouped into collections called 'packages'. A package defines the scope of the integers that denote the types specific to the classes in the package. Thus the integers shall be distinct. A package is denoted by an ASN.1 object identifier.

The closure of a package P is the set of classes that need to be supported in order to be able to create all possible instances of all classes defined in P.

Package closure is formally defined in terms of class closure, which is the set of classes that need to be supported in order to be able to create all possible instances of a particular class.

More specifically, the closure of a class C is a set that consists of:

1. the class C itself;
2. the closures of any subclasses of C defined in the same package as C, and
3. the closures of the classes of all permitted subobjects of instances of C.

The closure of a package P is the set of classes made up of the union of the closures of all the classes defined in P.

## 2.7 PACKAGE DEFINITIONS

For purposes of the generic interface, the definition of a package has the following elements:

1. The package's name, which denotes the package's object identifier.
2. The definitions of the one or more classes which make up the package.
3. The identification of zero or more concrete classes in the package to which the Create function applies (in every implementation of the service).
4. The identification of zero or more concrete classes in the package to which the Encode function applies (in every implementation of the service).
5. The explicit identification of the zero or more classes in other packages that appear in the package's closure (as a convenience to the reader).

## 2.8 WORKSPACES

Two application-specific APIs may involve the same class, the two APIs may employ different implementations of the service, for example, because they are supplied by different vendors, and the two implementations may represent private objects differently. If it is to use both application-specific APIs, the client must be able to specify which service implementation is to create an instance of the class that both support. In addition, the client may wish to present the object at *both* application-specific APIs, in which case the object must be converted from one internal format to another. Such interworking between service implementations is achieved by means of workspaces.

The service maintains private objects in workspaces. A 'workspace' is a repository for instances of classes in the closures of one or more packages associated with the workspace. The implementations of the OM interface functions may differ from one workspace to another. A package may be associated with any number of workspaces. The OM package is implicitly associated with every workspace. Other packages may be explicitly associated with a workspace when it is defined.

The interface includes functions for effectively copying and moving objects from one workspace to another, provided that the objects' classes are associated with both. How workspaces are created, made known to the client and destroyed, however, is outside the scope of this document. In all cases, destroying a workspace effectively applies the Delete function to each private object it contains.

*Note that typically workspaces are created, made known to the client, and destroyed by means of application-specific APIs designed to be used in conjunction with the present interface.*

*Note that failure to delete private objects before closing the workspace could result in consumption of resources by those objects with no mechanism available for freeing those resources.*

## 2.9 STORAGE MANAGEMENT

An object occupies storage. A public object occupies main storage, and a private object occupies main storage, secondary storage or a combination of the two, at the option of the workspace in which the object resides. The storage occupied by a public object is directly accessible to the client, while the storage occupied by a private object is not. The storage an object occupies is allocated and released by the client if the object is client-generated, or by the service if the object is service-generated or private.

An object is accessed via an object handle. An object handle is the means by which the client supplies an object to the service as an argument of an interface function, and the service returns an object as the result of an interface function to the client. For a public object, the object handle is simply a pointer to the data structure containing the object attributes. For a private object, the object handle is a pointer to a data structure whose layout is implementation-specific and is unknown to the client.

The client creates a client-generated public object by using normal programming language constructs. The client is responsible for managing any storage involved.

The service creates service-generated public objects and allocates any necessary storage. The client destroys a service-generated public object and releases the storage by applying the Delete function to it.

At any point in time, a private object is either accessible or inaccessible to the client. An object is accessible if the client possesses a valid object handle for it. The object is inaccessible otherwise, i.e. the client does not possess an object handle, or the handle is invalid. Should the client designate an inaccessible object as an argument, the effect on the service's subsequent behaviour is undefined.

The service makes a private object accessible by returning an object handle as the result of a function in this or another (application-specific) interface. The client makes such an object inaccessible by applying the Delete function to it, or by supplying it as an argument of any other function that, according to the specification, makes the argument inaccessible. Applying Delete to a service-generated public object does not make its private subobjects inaccessible. They can always be made accessible again by means of the Get function.

A private object is also destroyed when the workspace containing it is destroyed. A service-generated public object is unaffected by the destruction of the workspace that generated it. A client-generated public object is not associated with a workspace.

The storage occupied by a service-generated public object must not be changed by the client, and the effect of doing so is undefined. This includes all values (strings, subobjects, integers, etc.). It is possible, however, to use a value that is a private subobject as an argument to an interface function that modifies the subobject.



# Information Syntaxes

## 3.1 INTRODUCTION

This Chapter defines the permitted syntaxes of attribute values. The syntaxes are highly aligned with the types and type constructors of ASN.1. How a value of each syntax is represented in the C interface is specified by the Value data type (see **Section 4.2, Data Types**).

## 3.2 SYNTAX TEMPLATES

The names of certain syntaxes are constructed from syntax templates. A 'syntax template' is a lexical construct comprising a primary identifier followed by an asterisk enclosed in parentheses: *identifier* (\*).

A syntax template encompasses a group of related syntaxes. Any member of the group, without distinction, is denoted by the primary identifier alone: *identifier*. A particular member is denoted by the template with the asterisk replaced by one of a set of secondary identifiers associated with the template: *identifier*<sub>1</sub> (*identifier*<sub>2</sub>).

## 3.3 SYNTAXES

A variety of syntaxes are defined. Most are functionally equivalent to ASN.1 types, as documented in **Section 3.6, Relationship to ASN.1 Simple Types**, through to **Section 3.9, Relationship to ASN.1 Type Constructors**.

The following syntaxes are defined:

**Boolean** A value of this syntax is a Boolean, i.e., may be **false** or **true**.

**Enumeration (\*)**

A value of any syntax encompassed by this syntax template is one of a set of values associated with the syntax. The only significant characteristic of the values is that they are distinct.

The group of syntaxes encompassed by this template is open-ended. Zero or more members are added to the group by each package definition. The secondary identifiers that denote the members are assigned there also.

**Integer** A value of this syntax is a (positive or negative) integer.

**Null** The one value of this syntax is a valueless placeholder.

**Object (\*)** A value of any syntax encompassed by this syntax template is an object, any instance of a class associated with the syntax.

The group of syntaxes encompassed by this template is open-ended. One member is added to the group by each class definition. The secondary identifier that denotes the member is the name of the class.

**String (\*)** A value of any syntax encompassed by this syntax template is a string (as defined in **Section 3.4, Strings**) whose form and meaning are associated with the syntax.

The group of syntaxes encompassed by this template is closed. One syntax is defined for each ASN.1 string type. The secondary identifier that denotes the member is, in general, the first word of the type's name.

### 3.4 STRINGS

A 'string' is an ordered sequence of zero or more bits, octets or characters. A string is categorised as either a 'bit string', an 'octet string' or a 'character string' depending upon whether it comprises bits, octets or characters, respectively.

The value length of a string is the number of bits in a bit string, octets in an octet string, or characters in a character string. It is confined to the interval  $[0, 2^{32})$ . Any constraints on the value length of a string are specified in the appropriate class definitions.

The syntaxes that form the String group are identified in Table 2 below, which gives the secondary identifier assigned to each such syntax. The identifiers in the first, second and third columns denote the syntaxes of bit, octet and character strings, respectively. The String group comprises all syntaxes identified in the table.

Bit String Identifier	Octet String Identifier	Character String Identifier
Bit	Encoding <sup>1</sup>	General <sup>2</sup>
	Object Identifier <sup>3</sup>	Generalised Time <sup>2</sup>
	Octet	Graphic <sup>2</sup>
		IA5 <sup>2</sup>
		Numeric <sup>2</sup>
		Object Descriptor <sup>2</sup>
		Printable <sup>2</sup>
		Teletex <sup>2</sup>
		UTC Time <sup>2</sup>
		Videotex <sup>2</sup>
		Visible <sup>2</sup>

<sup>1</sup>The octets are those the BER permit for the contents octets of the encoding of a value of any ASN.1 type.

<sup>2</sup>The characters are those permitted by ASN.1's type of the same name. Values of these syntaxes are represented in their BER encoded form.

<sup>3</sup>The octets are those the BER permit for the contents octets of the encoding of a value of ASN.1's Object Identifier type.

**Table 2: String Syntax Identifiers**

### 3.5 REPRESENTATION OF STRING VALUES

In the service interface, a string value is represented by the string data type. This is defined in **Section 4.2.13, String**. The length of a string is the number of octets by which it is represented at the interface. It is confined to the interval  $[0, 2^{32})$ .

*Note that the length of a character string may not equal the number of characters it comprises, because for example a single character may be represented using several octets.*

When passing large string values across the interface it may be necessary to segment them. A segment is any zero or more contiguous octets of a string value. Segment boundaries are without semantic significance.

### 3.6 RELATIONSHIP TO ASN.1 SIMPLE TYPES

As shown in Table 3 below, for every ASN.1 simple type except Real, there is an OM syntax that is functionally equivalent to it. The simple types are listed in the first column of the table, and the corresponding syntax is in the second.

Type	Syntaxes
Bit String	String (Bit)
Boolean	Boolean
Integer	Integer
Null	Null
Object Identifier	String (Object Identifier)
Octet String	String (Octet)
Real	<i>none</i> <sup>1</sup>

<sup>1</sup>A future edition of this document may define a syntax corresponding to this type.

**Table 3: Syntax for ASN.1's Simple Types**

### 3.7 RELATIONSHIP TO ASN.1 USEFUL TYPES

As shown in Table 4 below, for every ASN.1 useful type, there is an OM syntax that is functionally equivalent to it. The useful types are listed in the first column of the table, and the corresponding syntaxes are in the second.

Type	Syntax
External	Object (External)
Generalised Time	String (Generalised Time)
Object Descriptor	String (Object Descriptor)
Universal Time	String (UTC Time)

**Table 4: Syntaxes for ASN.1's Useful Types**

### 3.8 RELATIONSHIP TO ASN.1 CHARACTER STRING TYPES

As shown in Table 5 below, for every ASN.1 character string type, there is an OM syntax that is functionally equivalent to it. The ASN.1 character string types are listed in the first column of the table, and the corresponding syntax in the second.

Type	Syntax
General String	String (General)
Graphic String	String (Graphic)
IA5 String	String (IA5)
-	String (Local)
Numeric String	String (Numeric)
Printable String	String (Printable)
Teletex String	String (Teletex)
Videotex String	String (Videotex)
Visible String	String (Visible)

**Table 5: Syntaxes for ASN.1's Character String Types**

### 3.9 RELATIONSHIP TO ASN.1 TYPE CONSTRUCTORS

As shown in Table 6 below, for some, but not all, ASN.1 type constructors there are functionally equivalent OM syntaxes. The constructors are listed in the first column of the table, and the corresponding syntaxes are in the second.

Type Constructor	Syntax
Any	String (Encoding)
Choice	Object <sup>1</sup>
Enumerated	Enumeration
Selection	<i>none</i> <sup>2</sup>
Sequence	Object <sup>1</sup>
Sequence Of	Object <sup>1</sup>
Set	Object <sup>1</sup>
Set Of	Object <sup>1</sup>
Tagged	<i>none</i> <sup>3</sup>

<sup>1</sup>See the text.

<sup>2</sup>This type constructor, a purely specification-time phenomenon, has no corresponding syntax.

<sup>3</sup>This type constructor is used to distinguish the alternatives of a choice or the elements of a sequence or set. This function is performed by attribute types, as indicated in the text.

**Table 6: Syntaxes for ASN.1's Type Constructors**

The effects of the principal type constructors may be achieved, in any of a variety of ways, by using objects to group attributes, or by using attributes to group values. An OM application designer may (but need not) model these constructors as classes of the following kinds:

**Choice** An attribute type may be defined for each alternative, exactly one being permitted in an instance of the class.



**Sequence**

An attribute type may be defined for each sequence or set element. If an element is OPTIONAL then the attribute should have zero or one values.

**Sequence Of or Set Of**

A single, multi-valued attribute may be defined.

An ASN.1 definition of an EnumeratedType component of a structured type is generally mapped to an OM attribute with an OM syntax **Enumeration (\*)** in this interface. Where the ASN.1 component is OPTIONAL this is generally indicated by an additional member of the enumeration rather than by the omission of the OM attribute. This leads to simpler programming in the application.



# *Service Interface*

## 4.1 INTRODUCTION

This Chapter defines the service interface. It specifies the functions that the service makes available to the client, the data types of which the arguments and results of those functions are data values, and the return codes that denote the outcomes (in particular, the exceptions) that the functions may report. This Chapter also summarises the declarations that define the C service interface.

## 4.2 DATA TYPES

This section defines, and Table 7 lists, the data types of the service interface. The data types of both the generic and C interfaces are specified. Those of the C interface are repeated in **Section 4.5, Declaration Summary**, which serves as a summary and a reference.

Data Type	Description
Boolean	type definition for a Boolean data value.
Descriptor	type definition for describing an attribute type and value.
Enumeration	type definition for an Enumerated data value.
Exclusions	type definition for 'exclusions' argument for the Get function.
Integer	type definition for an Integer data value.
Modifications	type definition for 'modifications' argument for the Put function.
Object	type definition for a handle to either a private or a public object.
Object Identifier	type definition for an Object Identifier data value.
Private Object	type definition for a handle to an object in an implementation-defined, or private, representation.
Public Object	type definition for a defined representation of an object that can be directly interrogated by a programmer.
Return Code	type definition for a value returned from all OM functions indicating either that the function succeeded or why it failed.
String	type definition for a data value of 'String' syntax.
Syntax	type definition for identifying a syntax type.
Type	type definition for identifying an OM attribute type.
Type List	type definition for enumerating a sequence of OM attribute types.
Value	type definition for representing any data value.
Value Position	type definition for designating a particular location within a String data value.
Workspace	type definition for identifying an application-specific API that implements OM, such as directory or message handling.

**Table 7: Service Interface Data Types**

Some data types are defined in terms of the following 'intermediate data types', whose precise definitions in C are system-defined:

**Sint** The positive and negative integers representable in 16 bits.

**Sint16** The positive and negative integers representable in 16 bits.

**Sint32** The positive and negative integers representable in 32 bits.

**Uint** The non-negative integers representable in 16 bits.

**Uint16** The non-negative integers representable in 16 bits.

**Uint32** The non-negative integers representable in 32 bits.

*Note that the Sint and Uint data types are defined above by the range of integers they must accommodate. As typically declared in the C interface, they are defined by the range of integers the host machine's word size permits. The latter range, however, always encompasses the former.*

**C DECLARATION**

```

typedef system-defined, e.g., int           OM_sint;
typedef system-defined, e.g., int           OM_sint16;
typedef system-defined, e.g., long int      OM_sint32;
typedef system-defined, e.g., unsigned     OM_uint;
typedef system-defined, e.g., unsigned     OM_uint16;
typedef system-defined, e.g., long unsigned OM_uint32;

```

**4.2.1 Boolean****NAME**

Boolean - type definition for a Boolean data value

**C DECLARATION**

```
typedef OM_uint32 OM_boolean;
```

**DESCRIPTION**

A data value of this data type is a Boolean, i.e. either **false** or **true**.

In the C interface, false is denoted by zero *{OM\_FALSE}*, true by any other integer, although the symbolic constant *{OM\_TRUE}* refers to the integer one specifically.

**4.2.2 Descriptor****NAME**

Descriptor - type definition for describing an attribute type and value

**C DECLARATION**

```

typedef struct OM_descriptor_struct
{
    OM_type      type;
    OM_syntax    syntax;
    OM_value     value;
} OM_descriptor;

```

*/\* Note that other components are encoded in high bits of the syntax member. \*/*

**DESCRIPTION**

A data value of this type is a descriptor, which embodies an attribute value. A sequence of descriptors (an array in C) can represent all the values of all the attributes of an object, and is the representation called a **Public Object**. A descriptor has the following components:

**Type (Type)** Identifies the type of the attribute value.

**Syntax (Syntax)**

Identifies the syntax of the attribute value.

In the C interface, **Long-String** to **Private** below are encoded in the high-order bits of this structure member. The syntax should always be masked with the constant *{OM\_S\_SYNTAX}* because of this. For example:

```

my_syntax = my_public_object[3].syntax & OM_S_SYNTAX;

my_public_object[4].syntax =
    my_syntax + (my_public_object[4].syntax & ~OM_S_SYNTAX);

```

**Long-String** (Boolean)

True, if and only, if the descriptor is service-generated and the length of the value is greater than an implementation-defined limit.

In the C interface, this component occupies bit 15 (0x8000) of the syntax and is represented by the constant `{OM_S_LONG_STRING}`.

**No-Value** (Boolean)

Only true if the descriptor is service-generated and the value is not present (because **exclude-values** or **exclude-multiples** was set in the call to `Get()`).

In the C interface, this component occupies bit 14 (0x4000) of the syntax and is represented by the constant `{OM_S_NO_VALUE}`.

**Local-String** (Boolean)

Only significant if the Syntax is `String(*)`. True if, and only if, the string is represented in an implementation-defined local character set. The local character set may be more amenable for use as keyboard input or display output than the non-local character set, and may include specific treatment of line termination sequences. Certain interface functions may convert information in string syntaxes to or from the local representation, which may result in a loss of information.

In the C interface, this component occupies bit 13 (0x2000) of the syntax and is represented by the constant `{OM_S_LOCAL_STRING}`.

**Service-Generated** (Boolean)

True if, and only if, the descriptor is service-generated and the first descriptor of a public object, or the defined part of a private object (see **Chapter 5, Workspace Interface**).

In the C interface, this component occupies bit 12 (0x1000) of the syntax and is represented by the constant `{OM_S_SERVICE_GENERATED}`.

**Private** (Boolean)

True if, and only if, the descriptor in the service-generated public object contains a reference to the handle of a private subobject, or in the defined part of a private object.

*Note that this applies only when the descriptor is service-generated. The client need not set this bit in a client-generated descriptor containing a reference to a private object.*

In the C interface, this component occupies bit 11 (0x0800) of the syntax and is represented by the constant `{OM_S_PRIVATE}`.

**Value (Value)**

The attribute value.

**4.2.3 Enumeration****NAME**

Enumeration - type definition for an Enumerated data value.

**C DECLARATION**

```
typedef OM_sint32 OM_enumeration;
```

**DESCRIPTION**

A data value of this data type is an attribute value whose syntax is an Enumeration syntax.

**4.2.4 Exclusions****NAME**

Exclusions - type definition for 'exclusions' argument of the Get function

**C DECLARATION**

```
typedef OM_uint OM_exclusions;
```

**DESCRIPTION**

A data value of this data type is an unordered set of one or more values, all of which are distinct. Each value denotes an exclusion, as defined by the Get function, and is chosen from the following set: **exclude-all-but-these-types**, **exclude-multiples**, **exclude-all-but-these-values**, **exclude-values**, **exclude-subobjects** and **exclude-descriptors**. Alternatively, the single value **no-exclusions** may be chosen which selects the entire object.

In the C interface, each value except **no-exclusions** is represented by a distinct bit, the presence of the value being represented as one, and its absence as zero. Thus multiple exclusions are requested by adding or, equivalently, or-ing the values that denote the individual exclusions.

**4.2.5 Integer****NAME**

Integer - type definition for an Integer data value

**C DECLARATION**

```
typedef OM_sint32 OM_integer;
```

**DESCRIPTION**

A data value of this data type is an attribute value whose syntax is Integer.

#### 4.2.6 Modification

**NAME**

Modification - type definition for 'modifications' argument of the Put function

**C DECLARATION**

```
typedef OM_uint OM_modification;
```

**DESCRIPTION**

A data value of this data type denotes a kind of modification, as defined by the Put function. It is chosen from the following set: **insert-at-beginning**, **insert-at-certain-point**, **insert-at-end**, **replace-all** and **replace-certain-values**.

#### 4.2.7 Object

**NAME**

Object - type definition for a handle to either a private or a public object

**C DECLARATION**

```
typedef struct OM_descriptor_struct *OM_object;
```

**DESCRIPTION**

A data value of this data type represents an object, public or private. It is an ordered sequence of one or more instances of the Descriptor data type. See the Private Object and Public Object data types for constraints upon that sequence.

#### 4.2.8 Object Identifier

**NAME**

Object Identifier - type definition for an Object Identifier data value

**C DECLARATION**

```
typedef OM_string OM_object_identifier;
```

**DESCRIPTION**

A data value of this data type contains an octet string that comprises the contents octets of the BER encoding of an ASN.1 object identifier.

**C Declaration of Object Identifiers**

Every application program that makes use of a class or other Object Identifier must explicitly import it into every compilation unit (C source module) that uses it. Each such class or Object Identifier name must be explicitly exported from just one compilation module. Most application programs will find it convenient to export all the names they use from the same compilation unit. Exporting and importing is done by the following two macros:

```
OM_IMPORT(class_name)
```

```
OM_IMPORT(OID_name)
```

This macro makes the class or other Object Identifier constants available within a compilation unit.



```
OM_EXPORT(class_name)
OM_EXPORT(OID_name)
```

This macro allocates memory for the constants that represent the class or other Object Identifier.

Package implementors must ensure that there are constants defined in the appropriate header files, with the define identifier having the prefix *OMP\_O\_* followed by the variable name for the object identifier. The constant itself provides the hexadecimal value of the object identifier string. See the example for *OMP\_O\_OM\_BER* (in **Section 4.5, Declaration Summary**).

### Use of Object Identifiers in C

```
OM_OID_DESC(type, OID_name)
```

This macro initialises a descriptor. It sets the type component to that given, sets the syntax component to *{OM\_S\_OBJECT\_IDENTIFIER\_STRING}*, and sets the value component to be the given Object Identifier.

```
OM_NULL_DESCRIPTOR
```

This macro initialises a descriptor to mark the end of a client-allocated public object.

```
OM_C_class_name
```

For each class, and for other Object Identifiers, there is a global variable of type *OM\_STRING* with the same name (e.g. the **External** class has a variable called *OM\_C\_EXTERNAL*; the Object Identifier for BER rules has a variable called *OM\_BER*). This variable can be supplied as an argument to functions when required. This variable is valid only when it is exported by an *OM\_EXPORT* macro, and imported by an *OM\_IMPORT* macro, in the compilation units that use it. This variable cannot form part of a descriptor, but the value of its length and elements components can be used.

```
/* Examples of the use of the macros and constants. */
```

```
#include <xom.h>
```

```
OM_IMPORT(OM_C_ENCODING)
OM_IMPORT(OM_CANONICAL_BER)
```

```
/* The following sequence must appear in exactly one compilation
 * unit in place of the above:
 *
 * #include <xom.h>
 *
 * OM_EXPORT(OM_C_ENCODING)
 * OM_EXPORT(OM_CANONICAL_BER)
 */
```

```

main()
{
/* Use #1 - Define a public object of class Encoding
 *   (Note that xxxx is a Message Handling class which can be
 *   encoded.)
 */
OM_descriptor my_public_object[] = {
    OM_OID_DESC(OM_CLASS, OM_C_ENCODING),
    OM_OID_DESC(OM_OBJECT_CLASS, MA_C_xxxx),
    { OM_OBJECT_ENCODING, OM_S_ENCODING, some_BER_value },
    OM_OID_DESC(OM_RULES, OM_CANONICAL_BER),
    OM_NULL_DESCRIPTOR
};

/* Use #2 - Pass class Encoding as an argument to om_instance()
 */
return_code = om_instance(my_object, OM_C_ENCODING, &boolean_result);
}

```

#### 4.2.9 Private Object

##### NAME

Private Object - type definition for a handle to an object in an implementation-defined, or private, representation

##### C DECLARATION

```
typedef OM_object OM_private_object;
```

##### DESCRIPTION

A data value of this data type is the designator or handle for a private object. It comprises a single descriptor whose Type component is **private-object** and whose Syntax and Value components are unspecified.

*Note that the descriptor's Syntax and Value components are essential to the service's proper operation with respect to the private object. Of no concern to the client, their nature is of concern to service implementors and is therefore further specified in **Chapter 5, Workspace Interface**. The service-generated {OM\_S\_SERVICE\_GENERATED} and private-object {OM\_S\_PRIVATE} bits in the syntax component are always set by the service, though they are of no concern to the client.*

#### 4.2.10 Public Object

##### NAME

Public Object - type definition for a defined representation of an object that can be directly interrogated by a programmer

##### C DECLARATION

```
typedef OM_object OM_public_object;
```

##### DESCRIPTION

A data value of this data type is a public object. It comprises one or (typically)

more descriptors, all but the last of which represent values of attributes of the object.

The descriptors for the values of a particular attribute having two or more values are adjacent to one another in the sequence. Their order is that of the values they represent. The order of the resulting groups of descriptors is unspecified.

To the extent that it is represented among the descriptors, the Class attribute specific to class Object shall be represented before any other attributes.

Whether or not the class attribute is present, the syntax field of the first descriptor must have the `{ OM_S_SERVICE_GENERATED }` bit set or cleared appropriately.

The last descriptor signals the end of the sequence of descriptors. Its Type component is **no-more-types**, and its Syntax component is **no-more-syntaxes**. The last descriptor's Value component is wholly unspecified.

#### 4.2.11 Return Code

##### NAME

Return Code - type definition for a value returned from all OM functions indicating either that the function succeeded or why it failed

##### C DECLARATION

```
typedef OM_uint OM_return_code;
```

##### DESCRIPTION

A data value of this data type is the integer in the interval  $[0, 2^{16})$  that denotes an outcome of an interface function. It is chosen from the set specified in **Section 4.3, Functions**.

All editions of this document employ integers in the (narrower) interval  $[0, 2^{15})$  to denote the return codes they define.

#### 4.2.12 String

##### NAME

String - type definition for a data value of String syntax

##### C DECLARATION

```
typedef OM_uint32 OM_string_length;

typedef struct {
    OM_string_length length;
    void *elements;
} OM_string;

#define OM_STRING(string)
    { (OM_string_length)(sizeof(string)-1), (string) }
```

##### DESCRIPTION

A data value of this data type is a string (i.e., an instance of a String syntax). In the C interface, a string is represented as either a length-specified or a null-terminated string. A string has the following components:

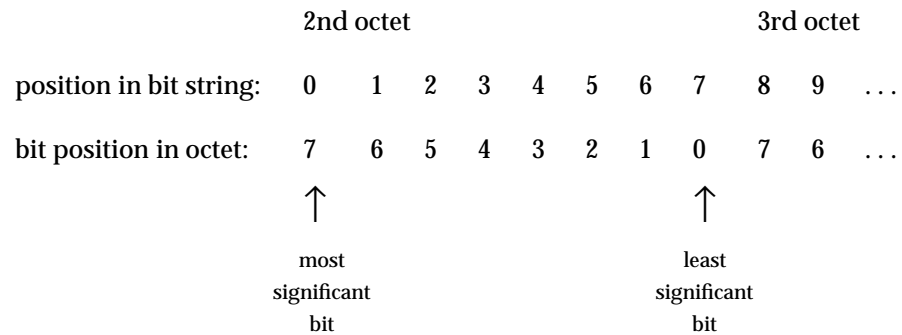
**Length** (String Length)

The number of octets by means of which the string is represented or the value **length-unspecified** if the string is null-terminated.

**Elements**

The string's elements (the octets that make up its value).

In the C interface, the bits of a bit string are represented as a sequence of octets as follows. The first octet stores the number of unused bits in the last octet. The bits in the bit string, commencing with the first bit and proceeding to the trailing bit, shall be placed in bits 7 to 0 of the second octet, followed by bits 7 to 0 of the third octet, followed by bits 7 to 0 of each octet in turn, followed by as many bits as are needed of the final octet, commencing with bit 7.



The service supplies a string value in the length-specified form. The client may supply a string value to the service in either form.

In the C interface, the characters of a character string are represented as any sequence of octets admissible as the primitive contents octets of the BER encoding of a value of the ASN.1 type that defines the variety of character string in question. A zero value character follows (and thus, depending upon the variety of character string, may delimit) the characters of the character string, but is not encompassed by the Length component.

In the C interface, a macro, *{OM\_STRING}*, is provided for fabricating a data value of this data type given only the value of its Elements component. The macro, however, applies to octet strings and character strings but not to bit strings.

**4.2.13 Syntax**

**NAME**

Syntax - type definition for identifying a syntax type

**C DECLARATION**

**typedef OM\_uint16 OM\_syntax;**

**DESCRIPTION**

A data value of this data type is an integer in the interval [0, 2<sup>9</sup>) that denotes an individual syntax, or a set of syntaxes taken together.

The data value is chosen from among the following:

1. **boolean**, **integer** and **null**. Each denotes the syntax of the same name.
2. **generalised-time-string**, **object-descriptor-string**, **object-identifier-string** and **utc-time-string**. Each denotes the String syntax of the same name.
3. **enumeration** and **object**. Each denotes a syntax associated with the syntax template of the same name.
4. **bit-string**, **encoding**, **general-string**, **graphic-string**, **ia5-string**, **numeric-string**, **octet-string**, **printable-string**, **teletex-string**, **videotex-string** and **visible-string**. Each denotes the String syntax of the same name.

All editions of this document employ integers in the (narrower) interval  $[0, 2^9)$  to denote the syntaxes they define. The range  $[2^9, 2^{10})$  is reserved for vendor extensions. Wherever possible, the integers used are the same as the corresponding ASN.1 universal class number. The exception to this rule is the uppermost integer available in the range defined by this document.

#### 4.2.14 Type

##### NAME

Type - type definition for identifying an OM attribute type.

##### C DECLARATION

```
typedef OM_uint16 OM_type;
```

##### DESCRIPTION

A data value of this data type is an integer in the interval  $[0, 2^{16})$  that denotes a type in the context of a package, except that the values **no-more-types**, and **private-object** have the meanings given them by the Type List and Private Object data types, respectively.

All editions of this document employ integers in the (narrower) interval  $[0, 2^{15})$  to denote the types they define.

#### 4.2.15 Type List

##### NAME

Type List - type definition for enumerating a sequence of OM attribute types

##### C DECLARATION

```
typedef OM_type *OM_type_list;
```

##### DESCRIPTION

A data value of this data type is an ordered sequence of zero or more type numbers, each an instance of the Type data type.

In the C interface, an additional data value, **no-more-types**, follows (and thus delimits) the sequence. The C representation of the sequence is an array.

#### 4.2.16 Value

**NAME**

Value - type definition for representing any data value

**C DECLARATION**

```
typedef struct {
    OM_uint32 padding;
    OM_object object;
} OM_padded_object;
```

*Note that the structure above, in particular, its Padding component, aligns the Object component with the Elements component of the String component below. This facilitates initialisation in C.*

```
typedef union OM_value_union {
    OM_string      string;
    OM_boolean     boolean;
    OM_enumeration enumeration;
    OM_integer     integer;
    OM_padded_object object;
} OM_value;
```

*Note that the identifier OM\_value\_union is defined for reasons of compilation order. It is used in the definition of the Descriptor data type.*

**DESCRIPTION**

A data value of this data type is an attribute value. It has no components if the value's syntax is **no-more-syntaxes**, or if the value's syntax is **no-value**, and (exactly) one of the following components otherwise:

**String** (String) The value if its syntax is a String syntax.

**Boolean** (Boolean)  
The value if its syntax is Boolean.

**Enumeration** (Enumeration)  
The value if its syntax is an Enumeration syntax.

**Integer** (Integer) The value if its syntax is Integer.

**Object** (Object) The value if its syntax is an Object syntax.

*Note that a data value of this data type appears only as a component of a descriptor. Thus it is always accompanied by indicators of the value's syntax. The latter indicator reveals which component above is present.*

#### 4.2.17 Value Length

**NAME**

Value Length - the number of bits, octets or characters in a string

**C DECLARATION**

```
typedef OM_uint32 OM_value_length;
```

**DESCRIPTION**

A data value of this data type is the number of bits in a bit string, octets in an octet string, or characters in a character string, an integer in the interval  $[0, 2^{32})$ .

*Note that this data type is not used in the definition of the interface. It is provided for use by client programmers in defining attribute constraints.*

**4.2.18 Value Position****NAME**

Value Position - type definition for denoting an attribute value's position within an attribute.

**C DECLARATION**

```
typedef OM_uint32 OM_value_position;
```

**DESCRIPTION**

A data value of this data type is the integer in the interval  $[0, 2^{32}-1)$  that denotes the position of a value within an attribute, except that the value **all-values** has the meaning given to it by the Get function.

**4.2.19 Workspace****NAME**

Workspace - type definition for identifying an application-specific API that implements OM, such as directory or message handling

**C DECLARATION**

```
typedef void *OM_workspace;
```

**DESCRIPTION**

A data value of this data type is the designator or handle for a workspace.

*Note - of no concern to the client, the nature of the handle is of concern to service implementors and is therefore further specified in **Chapter 5, Workspace Interface**.*

### 4.3 FUNCTIONS

This section defines, and Table 8 lists, the functions of the service interface. The functions of both the generic and C interfaces are specified. Those of the C interface are repeated in **Section 4.5, Declaration Summary**, which serves as a summary and a reference.

Function	Description
Copy	Copy a private object.
Copy Value	Copy a string between private objects.
Create	Create a private object.
Decode	Decode the result of encoding a private object.
Delete	Delete a private or service-generated object.
Encode	Encode a private object.
Get	Get copies of attribute values from a private object.
Instance	Test an object's class.
Put	Put attribute values into a private object.
Read	Read a segment of a string in a private object.
Remove	Remove attribute values from a private object.
Write	Write a segment of a string into a private object.

**Table 8: Service Interface Functions**

As indicated in the table, the service interface comprises a number of functions whose purpose and range of capabilities are summarised as follows:

1. **Copy.** This function creates an independent copy of an existing private object, and all of its subobjects. The copy is placed in the original's workspace, or in another specified by the client.
2. **Copy Value.** This function replaces an existing attribute value or inserts a new value in one private object with a copy of an existing attribute value found in another. Both values must be strings.
3. **Create.** This function creates a new private object that is an instance of a particular class. The object may be initialised with the attribute values specified as initial in the class' definition.

The service need not permit the client to explicitly create instances of all classes, but rather only those indicated, by a package's definition, as having this property.

4. **Decode.** This function creates a new private object that is an exact, but independent, copy of the object that an existing private object encodes. The encoding identifies the class of the existing object and the rules used to encode it. The allowed rules include, but are not limited to, the BER and the canonical BER.

The service need not permit the client to decode instances of all classes, but rather only those indicated, by a package's definition, as having this property.

5. **Delete.** This function deletes a service-generated public object, or makes a private object inaccessible.
6. **Encode.** This function creates a new private object that encodes an existing private object for conveyance between workspaces, transport via a network, or storage in a file. The client identifies the encoding rules that the service is to follow. The



allowed rules include, but are not limited to, the BER and the canonical BER.

The service need not permit the client to encode instances of all classes, but rather only those indicated, by a package's definition, as having this property.

7. **Get.** This function creates a new public object that is an exact, but independent, copy of an existing private object. The client may request certain exclusions, each of which reduces the copy to a portion of the original. The client may also request that values are converted from one syntax to another before they are returned.

The copy may exclude attributes of other than specified types, values at other than specified positions within an attribute, the values of multi-valued attributes, copies of (not handles for) subobjects, or all attribute values (revealing only an attribute's presence).

8. **Instance.** This function determines whether an object is an instance of a particular class. The client can determine an object's class simply by inspection. The utility of this function is that it reveals that an object is an instance of a particular class, even if the object is an instance of a subclass of that class.
9. **Put.** This function places or replaces, in one private object, copies of the attribute values of another public or private object.

The source values may be inserted before any existing destination values, before the value at a specified position in the destination attribute, or after any existing destination values. Alternatively, the source values may be substituted for any existing destination values or for the values at specified positions in the destination attribute.

10. **Read.** This function reads a segment of a value of an attribute of a private object. The value must be a string. The value may first be converted from one syntax to another. The function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.
11. **Remove.** This function removes and discards particular values of an attribute of a private object. The attribute itself is removed if no values remain.
12. **Write.** This function writes a segment of a value of an attribute to a private object. The value must be a string. The segment may first be converted from one syntax to another. The written segment is made the value's last, any elements beyond it being discarded. The function enables the client to write an arbitrarily long value without having to place a copy of the entire value in memory.

In the C interface, the functions are realised by macros. The function prototype in the C Synopsis clause of a function's specification is only an exposition aid.

The intent of the interface definition is that each function is atomic, i.e., that it either carries out its assigned task in full and reports success, or fails to carry out even a portion of the task and reports an exception. However, the service does not guarantee that a task will not occasionally be carried out in part but not in full.

Whether a function detects and reports each of the exceptions listed in the Errors clause of its specification is unspecified. If a function detects two or more exceptions, which it reports, is unspecified. If a function reports an exception for which a return code is defined, however, it uses that (rather than another) return code to do so.

## NAME

Copy - create a new private object that is an exact, but independent, copy of an existing private object

## SYNOPSIS

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_copy (  
    OM_private_object    original,  
    OM_workspace         workspace,  
    OM_private_object    *copy  
);
```

## DESCRIPTION

This function creates a new private object, the copy, that is an exact, but independent, copy of an existing private object, the original. The function is recursive in that copying the original also copies its subobjects.

## ARGUMENTS

**Original** (Private Object)

The original, which remains accessible.

**Workspace** (Workspace)

The workspace in which the copy is to be created. The original's class shall be in a package associated with this workspace.

## RESULTS

**Return Code** (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**Copy** (Private Object)

The copy. This result is present if, and only if, the Return Code result is **success**.

## ERRORS

function-interrupted, memory-insufficient, network-error, no-such-class, no-such-object, no-such-workspace, not-private, permanent-error, pointer-invalid, system-error, temporary-error or too-many-values.

**NAME**

Copy Value - place or replace a string in one private object with a copy of a string in another private object

**SYNOPSIS**

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_copy_value (
    OM_private_object    source,
    OM_type              source_type,
    OM_value_position   source_value_position,
    OM_private_object   destination,
    OM_type             destination_type,
    OM_value_position  destination_value_position
);
```

**DESCRIPTION**

This function places or replaces an attribute value in one private object, the destination, with a copy of an attribute value in another private object, the source. The source value shall be a string. The copy's syntax is that of the original.

**ARGUMENTS**

**Source** (Private Object)

The source, which remains accessible.

**Source Type** (Type)

Identifies the type of the attribute, one of whose values is to be copied.

**Source Value Position** (Value Position)

The position within the above attribute of the value to be copied.

**Destination** (Private Object)

The destination, which remains accessible.

**Destination Type** (Type)

Identifies the type of the attribute, one of whose values is to be placed or replaced.

**Destination Value Position** (Value Position)

The position within the above attribute of the value to be placed or replaced.

If the value Position exceeds the number of values present in the destination attribute, the argument is taken to be equal to that number.

**RESULTS**

**Return Code** (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**ERRORS**

function-declined, function-interrupted, memory-insufficient, network-error, no-such-object, no-such-type, not-present, not-private, permanent-error, pointer-

invalid, system-error, temporary-error, wrong-value-length, wrong-value-syntax or wrong-value-type.

**NAME**

Create - create a new private object that is an instance of a particular class

**SYNOPSIS**

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_create (
```

```
    OM_object_identifier    class,  
    OM_boolean            initialise,  
    OM_workspace         workspace,  
    OM_private_object    *object
```

```
);
```

**DESCRIPTION**

This function creates a new private object that is an instance of a particular class.

**ARGUMENTS****Class** (Object Identifier)

Identifies the class of the object to be created. The specified class shall be concrete.

**Initialise** (Boolean)

Whether the created object is to be initialised as specified in the definition of its class. If this argument is **true**, the object is made to comprise the attribute values specified as initial values in the tabular definitions of the object's class and its superclasses. If this argument is **false**, the object is made to comprise the Class attribute alone.

*Note that by subsequently adding new values to the object and replacing and removing existing values, the client can create all conceivable instances of the object's class.*

**Workspace** (Workspace)

The workspace in which the object is to be created. The specified class shall be in a package associated with this workspace.

**RESULTS****Return Code** (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**Object** (Private Object)

The created object. This result is present if, and only if, the Return Code result is **success**.

**ERRORS**

function-declined, function-interrupted, memory-insufficient, network-error, no-such-class, no-such-workspace, not-concrete, permanent-error, pointer-invalid, system-error or temporary-error.

## NAME

Decode - create a new private object that is an exact, but independent, copy of the object that an existing private object encodes

## SYNOPSIS

```
[#include <xom.h>]
```

```
OM_return_code  
om_decode (  
    OM_private_object      encoding,  
    OM_private_object      *original  
);
```

## DESCRIPTION

This function creates a new private object, the original, that is an exact, but independent, copy of the object that an existing private object, the encoding, encodes.

## ARGUMENTS

### Encoding (Private Object)

The encoding, which remains accessible. It shall be an instance of class Encoding.

## RESULTS

### Return Code (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

### Original (Private Object)

The original, which is created in the encoding's workspace. This result is present if, and only if, the Return Code result is **success**.

## ERRORS

encoding-invalid, function-interrupted, memory-insufficient, network-error, no-such-class, no-such-object, no-such-rules, not-an-encoding, not-private, permanent-error, pointer-invalid, system-error, temporary-error, too-many-values, wrong-value-length, wrong-value-makeup, wrong-value-number, wrong-value-syntax or wrong-value-type.

**NAME**

Delete - delete a private or service-generated object

**SYNOPSIS**

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_delete (  
    OM_object      subject  
);
```

**DESCRIPTION**

This function deletes a service-generated public object, or makes a private object inaccessible. It is not intended for use on client-generated public objects.

If applied to a service-generated public object, the function deletes the object and releases any resources associated with the object, including the space occupied by descriptors and attribute values. The function is applied recursively to any public subobjects. There is no effect on any private subobjects. The user should not use **om\_delete** directly on public subobjects (e.g. subobjects existing within the public object 'copy' returned from a call to **om\_get** with no exclusions), but should only apply the function to the top-level object (the object pointed to by 'copy' itself), otherwise results are unspecified.

If applied to a private object, the function makes the object inaccessible. Any existing object handles for the object are invalidated. The function is applied recursively to any private subobjects.

**ARGUMENTS**

**Subject** (Object)

The subject, which is to be deleted.

**RESULTS**

**Return Code** (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**ERRORS**

function-interrupted, memory-insufficient, network-error, no-such-object, no-such-syntax, no-such-type, not-the-services, permanent-error, pointer-invalid, system-error or temporary-error.

## NAME

Encode - create a new private object that encodes an existing private object

## SYNOPSIS

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_encode (
```

```
    OM_private_object      original,
```

```
    OM_object_identifier rules,
```

```
    OM_private_object      *encoding
```

```
);
```

## DESCRIPTION

This function creates a new private object, the encoding, that exactly and independently encodes an existing private object, the original. The client identifies the set of rules that the service is to follow to produce the encoding.

The definition of a package identifies zero or more of its concrete classes to which this function applies. Thus the function will encode instances of those classes. It will also encode instances of zero or more additional concrete classes in the package. The identities of these latter classes are implementation-defined.

## ARGUMENTS

**Original** (Private Object)

The original, which remains accessible.

**Rules** (Object Identifier)

Identifies the set of rules that the service is to follow to produce the encoding. The defined values of this argument are those of the Rules attribute specific to the Encoding class.

## RESULTS

**Return Code** (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**Encoding** (Private Object)

The encoding, an instance of class Encoding, which is created in the original's workspace. This result is present if, and only if, the Return Code result is **success**.

## ERRORS

function-declined, function-interrupted, memory-insufficient, network-error, no-such-object, no-such-rules, not-private, permanent-error, pointer-invalid, system-error or temporary-error.



**NAME**

Get - create a public copy of all or particular parts of a private object

**SYNOPSIS**

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_get (
    OM_private_object      original,
    OM_exclusions         exclusions,
    OM_type_list         included_types,
    OM_boolean          local_strings,
    OM_value_position    initial_value,
    OM_value_position    limiting_value,
    OM_public_object     *copy,
    OM_value_position    *total_number
);
```

**DESCRIPTION**

This function creates a new public object, the copy, that is an exact, but independent, copy of an existing private object, the original. The client may request certain exclusions, each of which reduces the copy to a portion of the original.

One exclusion is always requested implicitly. For each attribute value in the original that is a string whose length exceeds an implementation-defined number, the copy includes a descriptor that omits the elements (but not the length) of the string; the Elements component of the String component of the Value component of the descriptor is **elements-unspecified**, and the **Long-String** bit of the Syntax component is set to **true**.

*Note that the client can access long values by means of the Read function.*

**ARGUMENTS****Original** (Private Object)

The original, which remains accessible.

**Exclusions** (Exclusions)

Explicit requests for zero or more exclusions, each of which reduces the copy to a prescribed portion of the original. The exclusions apply to the attributes of the object but not to those of its subobjects.

Apart from **no-exclusions**, each value is chosen from the following list. When multiple exclusions are specified each is applied in the order in which it appears in the list, with lower numbered exclusions having precedence over higher numbered exclusions. If, after the application of an exclusion, that portion of the object would not be returned, no further exclusions need be applied to that portion.

1. **exclude-all-but-these-types**. The copy includes descriptors encompassing only attributes of specified types.

*Note that this exclusion provides a means for determining the values of specified attributes, as well as the syntaxes of those values.*

2. **exclude-multiples.** The copy includes a single descriptor for each attribute (actually) having two or more values, rather than one descriptor for each value. Each such descriptor contains no attribute value and the **No-Value** bit of the syntax component is set.

If the attribute (actually) has values of two or more syntaxes, the descriptor identifies one of those syntaxes, but which one is unspecified.

*Note that this exclusion provides a means for discerning the presence of multi-valued attributes without simultaneously getting their values.*

3. **exclude-all-but-these-values.** The copy includes descriptors encompassing only values at specified positions within an attribute.

*Note that when used in conjunction with the **exclude-all-but-these-types** exclusion, this exclusion provides a means for determining the values of a specified attribute, as well as the syntaxes of those values, one or more but not all attributes at a time.*

4. **exclude-values.** The copy includes a single descriptor for each attribute value, but the descriptor does not contain the value, and the **No-Value** bit of the syntax component is set.

*Note that this exclusion provides a means for determining an objects composition, i.e., the type and syntax of each of its attribute values.*

5. **exclude-subobjects.** The copy includes, for each value whose syntax is object, a descriptor containing an object handle for the original private subobject, rather than a public copy of it. This handle thus makes that subobject accessible for use in subsequent function calls.

*Note that this exclusion provides a means for examining an object one 'level' at a time.*

6. **exclude-descriptors.** When this exclusion is specified, no descriptors are returned and the copy result is not present. The Total Number result reflects the number of descriptors that would have been returned by applying the other inclusion and exclusion specifications.

*Note that this exclusion provides an attribute analysis capability. For instance, the total number of values in a multi-valued attribute can be determined by specifying an inclusion of the specific attribute type, and exclusions of **exclude-all-but-these-types**, **exclude-subobjects** and **exclude-descriptors**.*

*Note that the **exclude-all-but-these-values** exclusion affects the choice of descriptors, while the **exclude-values** exclusion affects the composition of descriptors.*

**Included Types** (Type List)

Present if, and only if, the **exclude-all-but-these-types** exclusion is requested, and identifies the types of the attributes to be included in the copy (provided that they appear in the original).

**Local Strings** (Boolean)

If true, indicates that all String(\*) values included in the copy are to be translated into the implementation-defined local character set representation (which may entail the loss of some information).

**Initial Value** (Value Position)

Present if, and only if, the **exclude-all-but-these-values** exclusion is requested, the position within each attribute of the first value to be included in the copy.

If it is **all-values** or exceeds the number of values present in an attribute, the argument is taken to be equal to that number.

**Limiting Value** (Value Position)

Present if, and only if, the **exclude-all-but-these-values** exclusion is requested, the position within each attribute one beyond that of the last value to be included in the copy. If this argument is not greater than the Initial Value argument, no values are included (and hence no descriptors are returned).

If it is **all-values** or exceeds the number of values present in an attribute, the argument is taken to be equal to that number.

**RESULTS****Return Code** (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**Copy** (Public Object)

The copy. This result is present if, and only if, the Return Code result is **success** and the **exclude-descriptors** exclusion is not specified.

The space occupied by the public object, and every attribute value that is a string, is service-provided. If the client alters any portion of that space, the effect upon the service's subsequent behavior is unspecified.

**Total Number** (Value Position)

The number of attribute descriptors returned in the public object, but not in any of its subobjects, based on the inclusion and exclusion arguments specified. If the **exclude-descriptors** exclusion is specified no Copy result is returned and the Total Number result reflects the actual number of attribute descriptors that would have been returned based on the remaining inclusion and exclusion values.

*Note that the total includes only the attribute descriptors in the Copy result. It excludes the special descriptor signalling the end of a public object.*

## **ERRORS**

function-interrupted, memory-insufficient, network-error, no-such-exclusion, no-such-object, no-such-type, not-private, permanent-error, pointer-invalid, system-error, temporary-error, wrong-value-syntax or wrong-value-type.

**NAME**

Instance - determine whether an object is an instance of a particular class or any of its subclasses

**SYNOPSIS**

```
[#include <xom.h>]
```

```
OM_return_code
om_instance (
    OM_object           subject,
    OM_object_identifier class,
    OM_boolean         *instance
);
```

**DESCRIPTION**

This function determines whether a service-generated public or private object, the subject, is an instance of a particular class or any of its subclasses.

*Note that the client can determine an object's class, C, by simply inspecting the object (using programming language constructs if the object is public, or the Get function if it is private). The utility of the present function is that it reveals that an object is an instance of the specified class, even if C is a subclass of that class.*

**ARGUMENTS****Subject** (Object)

The subject, which remains accessible.

**Class** (Object Identifier)

Identifies the class in question.

**RESULTS****Return Code** (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**Instance** (Boolean)

Whether the subject is an instance of the specified class or any of its subclasses. This result is present if, and only if, the Return Code result is **success**.

**ERRORS**

function-interrupted, memory-insufficient, network-error, no-such-class, no-such-object, no-such-syntax, not-the-services, permanent-error, pointer-invalid, system-error or temporary-error.

**NAME**

Put - place or replace in one private object copies of the attribute values of another, public or private object

**SYNOPSIS**

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_put (
    OM_private_object      destination,
    OM_modification       modification,
    OM_object             source,
    OM_type_list          included_types,
    OM_value_position     initial_value,
    OM_value_position     limiting_value
);
```

**DESCRIPTION**

This function places or replaces in one private object, the destination, copies of the attribute values of another, public or private object, the source. The client may specify that the source's values are to replace all or particular values in the destination, or to be inserted at a particular position within each attribute. All string values being copied that are in the local representation are first converted into the non-local representation for that syntax (which may entail the loss of some information).

**ARGUMENTS****Destination** (Private Object)

The destination, which remains accessible. The destination's class is unaffected.

**Modification** (Modification)

The nature of the requested modification. The modification determines how the Put function uses the attribute values in the source to modify the object. In all cases, for each attribute present in the source, copies of its values are placed in the object's destination attribute of the same type. The data value is chosen from among the following:

1. **insert-at-beginning.** The source values are inserted before any existing destination values. (The latter are retained.)
2. **insert-at-certain-point.** The source values are inserted before the value at a specified position in the destination attribute. (The latter are retained.)
3. **insert-at-end.** The source values are inserted after any existing destination values. (The latter are retained.)
4. **replace-all.** The source values are placed in the destination attribute. The existing destination values, if any, are discarded. (The latter are discarded.)

5. **replace-certain-values.** The source values are substituted for the values at specified positions in the destination attribute. (The latter are discarded.)

**Source (Object)**

The source, which remains accessible. The source's class is ignored. However, the attributes being copied from the source must be compatible with the destination's class definition.

**Included Types (Type List)**

If present, identifies the types of the attributes to be included in the destination (provided that they appear in the source); otherwise all attributes are to be included.

**Initial Value (Value Position)**

Present if, and only if, the Modification argument is **insert-at-certain-point** or **replace-certain-values**, the position within each destination attribute at which source values are to be inserted, or of the first value to be replaced, respectively.

If it is **all-values** or exceeds the number of values present in a destination attribute, the argument is taken to be equal to that number.

**Limiting Value (Value Position)**

Present if, and only if, the Modification argument is **replace-certain-values**, the position within each destination attribute, one beyond that of the last value to be replaced. If this argument is present, it must be greater than the Initial Value argument.

If it is **all-values** or exceeds the number of values present in a destination attribute, the argument is taken to be equal to that number.

**RESULTS****Return Code (Return Code)**

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**ERRORS**

function-declined, function-interrupted, memory-insufficient, network-error, no-such-class, no-such-modification, no-such-object, no-such-syntax, no-such-type, not-concrete, not-present, not-private, permanent-error, pointer-invalid, system-error, temporary-error, too-many-values, values-not-adjacent, wrong-value-length, wrong-value-makeup, wrong-value-number, wrong-value-position, wrong-value-syntax or wrong-value-type.

## NAME

Read - read a segment of a string in a private object

## SYNOPSIS

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_read (  
    OM_private_object    subject,  
    OM_type              type,  
    OM_value_position    value_position,  
    OM_boolean          local_string,  
    OM_string_length    *string_offset,  
    OM_string           *elements  
);
```

## DESCRIPTION

This function reads a segment of an attribute value in a private object, the subject. The segment that is returned is a segment of the string value that would have been returned if the complete value had been read in a single call.

*Note that this function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.*

## ARGUMENTS

### **Subject** (Private Object)

The subject, which remains accessible.

**Type** (Type) Identifies the type of the attribute, one of whose values is to be read.

### **Value Position** (Value Position)

The position within the above attribute of the value to be read.

### **Local-String** (Boolean)

If true, indicates that the value is to be translated into the implementation-defined local character set representation (which may entail the loss of some information).

### **Starting Position** (String Offset)

The offset, in octets, of the start of the string segment to be read.

If it exceeds the total length of the string, the argument is taken to be equal to the string length.

In the C interface, the Starting Position argument and the Next Position result of the generic interface are realised as the String Offset argument.



**Elements (String)**

The space the client provides for the segment to be read. The string's contents initially are unspecified. The string's length initially is the number of octets required to contain the segment that the function is to read.

The service modifies this argument. The string's elements are made the elements actually read. The string's length is made the number of octets required to hold the segment actually read. This may be smaller than the initial length if the segment is the last in a long string.

If **Local-String** is **true**, the segments that will be returned will be those of the translated string. Depending on the characteristics of the implementation-defined local character set, these may not correspond directly to the segments that would be obtained if **Local-String** were **false**.

**RESULTS****Return Code (Return Code)**

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

**Next Position (String Offset)**

The offset, in octets, of the start of the next string segment to be read, or zero if the value's final segment was read. This result is present if, and only if, the Return Code result is **success**.

In the C interface, the Starting Position argument and the Next Position result of the generic interface are realised as the String Offset argument. The value returned as the Next Position result may be used as the value for the Starting Position argument in the next call of the function. This allows for sequential reading of the value of a long string.

**ERRORS**

function-interrupted, memory-insufficient, network-error, no-such-object, no-such-type, not-present, not-private, permanent-error, pointer-invalid, system-error, temporary-error or wrong-value-syntax.

## NAME

Remove - remove and discard values of an attribute of a private object

## SYNOPSIS

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_remove (  
    OM_private_object    subject,  
    OM_type              type,  
    OM_value_position    initial_value,  
    OM_value_position    limiting_value  
);
```

## DESCRIPTION

This function removes and discards particular values of an attribute of a private object, the subject. If no values remain, the attribute itself is removed also. If the value is a subobject, the value is first removed and then the Delete function is applied to it, thus destroying the object.

## ARGUMENTS

### Subject (Private Object)

The subject, which remains accessible. The subject's class is unaffected.

### Type (Type)

Identifies the type of the attribute, some of whose values are to be removed. The type shall not be Class.

### Initial Value (Value Position)

The position within the above attribute of the first value to be removed.

If it is **all-values**, or exceeds the number of values present in the attribute, the argument is taken to be equal to that number.

### Limiting Value (Value Position)

The position within the attribute one beyond that of the last value to be removed. If this argument is not greater than the Initial Value argument, no values are removed.

If it is **all-values** or exceeds the number of values present in an attribute, the argument is taken to be equal to that number.

## RESULTS

### Return Code (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

## ERRORS

function-declined, function-interrupted, memory-insufficient, network-error, no-such-object, no-such-type, not-private, permanent-error, pointer-invalid, system-error or temporary-error.

**NAME**

Write - write a segment of a string into a private object

**SYNOPSIS**

```
[#include <xom.h>]
```

```
OM_return_code
```

```
om_write (
    OM_private_object      subject,
    OM_type                type,
    OM_value_position     value_position,
    OM_syntax             syntax,
    OM_string_length      *string_offset,
    OM_string             elements
);
```

**DESCRIPTION**

This function writes a segment of an attribute value in a private object, the subject. The segment that is supplied is a segment of the string value that would have been supplied if the complete value had been written in a single call.

The written segment is made the value's last; the function discards any values whose offset equals or exceeds the Starting Position argument. If the value being written is in the local representation, it is converted to the non-local representation (which may entail the loss of information and which may yield a different number of elements than that provided).

*Note that this function enables the client to write an arbitrarily long value without having to place a copy of the entire value in memory.*

**ARGUMENTS****Subject** (Private Object)

The subject, which remains accessible.

**Type** (Type)

Identifies the type of the attribute, one of whose values is to be written.

**Value Position** (Value Position)

The position within the above attribute of the value to be written. The value position shall neither be negative nor exceed the number of values present. If it equals the number of values present, the segment is inserted into the attribute as a new value.

**Syntax** (Syntax)

If the value being written was not already present in the subject, this identifies the syntax the value is to have. It must be a permissible syntax for the attribute of which this is a value. If the value being written was already present in the subject then that value's syntax is preserved and this argument is ignored.

## **Starting Position** (String Offset)

The offset, in octets, of the start of the string segment to write.

If it exceeds the current length of the string value being written, the argument is taken to be equal to that current length.

In the C interface, the Starting Position argument and the Next Position result of the generic interface are realised as the String Offset argument.

## **Elements** (String)

The string segment to be written. A copy of this segment will occupy a position within the string value being written, starting at the offset given by the Starting Position argument. Any values already at or beyond this offset are discarded.

## **RESULTS**

### **Return Code** (Return Code)

Whether the function succeeded and, if not, why. It is **success** or one of the values listed under Errors below.

### **Next Position** (String Offset)

The offset, in octets, after the last string segment written. This result is present if, and only if, the Return Code result is **success**.

In the C interface, the Starting Position argument and the Next Position result of the generic interface are realised as the String Offset argument. The value returned as the Next Position result may be used as the value for the Starting Position argument in the next call of the function. This allows for sequential writing of the value of a long string.

## **ERRORS**

function-declined, function-interrupted, memory-insufficient, network-error, no-such-object, no-such-syntax, no-such-type, not-present, not-private, permanent-error, pointer-invalid, system-error, temporary-error, wrong-value-length, wrong-value-makeup, wrong-value-position or wrong-value-syntax.

#### 4.4 RETURN CODES

This section defines, and Table 9 lists, the return codes of the service interface, and thus the exceptions that can prevent the successful completion of an interface function. The return codes of the generic interface alone are specified here. The return codes of the C interface are specified in **Section 4.5, Declaration Summary**.

The table's first column lists the return codes. The other columns identify with an "x" the return codes that apply to each function. (This information, organised differently, also appears in the Errors clauses of the function descriptions of **Section 4.3, Functions**.)

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
encoding-invalid	-	-	-	X	-	-	-	-	-	-	-	-
function-declined	-	X	X	-	-	X	-	-	X	-	X	X
function-interrupted	X	X	X	X	X	X	X	X	X	X	X	X
memory-insufficient	X	X	X	X	X	X	X	X	X	X	X	X
network-error	X	X	X	X	X	X	X	X	X	X	X	X
no-such-class	X	-	X	X	-	-	-	X	X	-	-	-
no-such-exclusion	-	-	-	-	-	-	X	-	-	-	-	-
no-such-modification	-	-	-	-	-	-	-	-	X	-	-	-
no-such-object	X	X	-	X	X	X	X	X	X	X	X	X
no-such-rules	-	-	-	X	-	X	-	-	-	-	-	-
no-such-syntax	-	-	-	-	X	-	-	X	X	-	-	X
no-such-type	-	X	-	-	X	-	X	-	X	X	X	X
no-such-workspace	X	-	X	-	-	-	-	-	-	-	-	-
not-an-encoding	-	-	-	X	-	-	-	-	-	-	-	-
not-concrete	-	-	X	-	-	-	-	-	X	-	-	-
not-present	-	X	-	-	-	-	-	-	X	X	-	X
not-private	X	X	-	X	-	X	X	-	X	X	X	X
not-the-services	-	-	-	-	X	-	-	X	-	-	-	-
permanent-error	X	X	X	X	X	X	X	X	X	X	X	X
pointer-invalid	X	X	X	X	X	X	X	X	X	X	X	X
success	X	X	X	X	X	X	X	X	X	X	X	X
system-error	X	X	X	X	X	X	X	X	X	X	X	X
temporary-error	X	X	X	X	X	X	X	X	X	X	X	X
too-many-values	X	-	-	X	-	-	-	-	X	-	-	-
values-not-adjacent	-	-	-	-	-	-	-	-	X	-	-	-
wrong-value-length	-	X	-	X	-	-	-	-	X	-	-	X
wrong-value-makeup	-	-	-	X	-	-	-	-	X	-	-	X
wrong-value-number	-	-	-	X	-	-	-	-	X	-	-	-
wrong-value-position	-	-	-	-	-	-	-	-	X	-	-	X
wrong-value-syntax	-	X	-	X	-	-	X	-	X	X	-	X
wrong-value-type	-	X	-	X	-	-	X	-	X	-	-	-

**Table 9: Service Interface Return Codes**

The return codes are as follows:

**success**           The function completed successfully.

**encoding-invalid**

The octets that constitute the value of an encoding's Object Encoding attribute are invalid.

**function-declined**

The function does not apply to the object to which it is addressed.

**function-interrupted**

The function was aborted by an external force (e.g. a keystroke, designated for this purpose, at a user interface).

<b>memory-insufficient</b>	The service cannot allocate the main memory it needs to complete the function.
<b>network-error</b>	The service could not successfully employ the network upon which its implementation depends.
<b>no-such-class</b>	A purported class identifier is undefined.
<b>no-such-exclusion</b>	A purported exclusion identifier is undefined.
<b>no-such-modification</b>	A purported modification identifier is undefined.
<b>no-such-object</b>	A purported object is nonexistent or the purported handle is invalid.
<b>no-such-rules</b>	A purported rules identifier is undefined.
<b>no-such-syntax</b>	A purported syntax identifier is undefined.
<b>no-such-type</b>	A purported type identifier is undefined.
<b>no-such-workspace</b>	A purported workspace is nonexistent.
<b>not-an-encoding</b>	An object is not an instance of the Encoding class.
<b>not-concrete</b>	A class is abstract, not concrete.
<b>not-present</b>	An attribute value is absent, not present.
<b>not-private</b>	An object is public, not private.
<b>not-the-services</b>	An object is client-generated, rather than service-generated or private.
<b>permanent-error</b>	The service encountered a permanent difficulty other than those denoted by other return codes.
<b>pointer-invalid</b>	In the C interface, an invalid pointer was supplied as a function argument or as the receptacle for a function result.
<b>system-error</b>	The service could not successfully employ the operating system upon which its implementation depends.
<b>temporary-error</b>	The service encountered a temporary difficulty other than those denoted by other return codes.
<b>too-many-values</b>	An implementation limit prevents the addition to an object of another attribute value. This limit is undefined.
<b>values-not-adjacent</b>	The descriptors for the values of a particular attribute are not adjacent.

**wrong-value-length**

An attribute has, or would have, a value that violates the value length constraints in force.

**wrong-value-makeup**

An attribute has, or would have, a value that violates a constraint of the value's syntax.

**wrong-value-number**

An attribute has, or would have, a value that violates the value number constraints in force.

**wrong-value-position**

The usage of value position(s) identified in the argument(s) of a function is invalid.

**wrong-value-syntax**

An attribute has, or would have, a value whose syntax is not permitted.

**wrong-value-type**

An object has, or would have, an attribute whose type is not permitted.

#### 4.5 DECLARATION SUMMARY

This section lists the declarations that define the C service interface. All of the declarations, except those for symbolic constants, also appear in **Section 4.2, Data Types** and **Section 4.3, Functions**.

The function macros that appear (in **Section 5.3, Types and Macros**) in the specification of the workspace interface here replace the function prototypes that appear (in **Section 4.3, Functions**) in the specification of the service interface.

The declarations, as assembled here, constitute the contents of a header file to be made accessible to client programmers. The header file includes by reference a second header file comprising the declarations defining the C workspace interface (see **Chapter 5, Workspace Interface**). The header files are `<xom.h>` and `<xomi.h>`, respectively. The symbols the declarations define are the only symbols the service makes visible to the client.

```

/* BEGIN SERVICE INTERFACE */

/* INTERMEDIATE DATA TYPES */

typedef system-defined, e.g., int          OM_sint;
typedef system-defined, e.g., int          OM_sint16;
typedef system-defined, e.g., long int     OM_sint32;
typedef system-defined, e.g., unsigned    OM_uint;
typedef system-defined, e.g., unsigned    OM_uint16;
typedef system-defined, e.g., long unsigned OM_uint32;

/* PRIMARY DATA TYPES */

/* Boolean */
typedef OM_uint32 OM_boolean;

/* String Length */
typedef OM_uint32 OM_string_length

/* Enumeration */
typedef OM_sint32 OM_enumeration;

/* Exclusions */
typedef OM_uint OM_exclusions;

/* Integer */
typedef OM_sint32 OM_integer;

/* Modification */
typedef OM_uint OM_modification;

```



```
/* Object */
typedef struct OM_descriptor_struct *OM_object;

/* String */
typedef struct {
    OM_string_length length;
    void *elements;
} OM_string;

#define OM_STRING(string)
    { (OM_string_length)(sizeof(string)-1), (string) }

/* Workspace */
typedef void *OM_workspace;

/* SECONDARY DATA TYPES */

/* Object Identifier */
typedef OM_string OM_object_identifier;

/* Private Object */
typedef OM_object OM_private_object;

/* Public Object */
typedef OM_object OM_public_object;

/* Return Code */
typedef OM_uint OM_return_code;

/* Syntax */
typedef OM_uint16 OM_syntax;

/* Type */
typedef OM_uint16 OM_type;
```

```

/* Type List */
typedef OM_type *OM_type_list;

/* Value */
typedef struct {
    OM_uint32    padding;
    OM_object    object;
} OM_padded_object;

typedef union OM_value_union {
    OM_string    string;
    OM_boolean   boolean;
    OM_enumeration enumeration;
    OM_integer   integer;
    OM_padded_object object;
} OM_value;

/* Value Length */
typedef OM_uint32 OM_value_length;

/* Value Position */
typedef OM_uint32 OM_value_position;

/* TERTIARY DATA TYPES */

/* Descriptor */
typedef struct OM_descriptor_struct {
    OM_type        type;
    OM_syntax      syntax;
    union OM_value_union value;
} OM_descriptor;

/* SYMBOLIC CONSTANTS */

/* Boolean */
#define OM_FALSE    ( (OM_boolean) 0 )
#define OM_TRUE     ( (OM_boolean) 1 )

/* Element Position */
#define OM_LENGTH_UNSPECIFIED    ( (OM_string_length) 0xFFFFFFFF )

```

```

/* Exclusions */

#define OM_NO_EXCLUSIONS                ( (OM_exclusions) 0 )
#define OM_EXCLUDE_ALL_BUT_THESE_TYPES   ( (OM_exclusions) 1 )
#define OM_EXCLUDE_ALL_BUT_THESE_VALUES  ( (OM_exclusions) 2 )
#define OM_EXCLUDE_MULTIPLES             ( (OM_exclusions) 4 )
#define OM_EXCLUDE_SUBOBJECTS            ( (OM_exclusions) 8 )
#define OM_EXCLUDE_VALUES                 ( (OM_exclusions) 16 )
#define OM_EXCLUDE_DESCRIPTOR            ( (OM_exclusions) 32 )

/* Modification */

#define OM_INSERT_AT_BEGINNING            ( (OM_modification) 1 )
#define OM_INSERT_AT_CERTAIN_POINT        ( (OM_modification) 2 )
#define OM_INSERT_AT_END                  ( (OM_modification) 3 )
#define OM_REPLACE_ALL                    ( (OM_modification) 4 )
#define OM_REPLACE_CERTAIN_VALUES         ( (OM_modification) 5 )

/* Object Identifiers */

/* NOTE: These macros rely on the ## token-pasting operator of ANSI C.
 * On many pre-ANSI compilers the same effect can be obtained by
 * replacing ## with /*. */

/* Private macro to calculate length of an object identifier.
 */
#define OMP_LENGTH(oid_string) (sizeof(OMP_O_##oid_string)-1)

/* Macro to initialise the syntax and value of an object identifier.
 */
#define OM_OID_DESC(type, oid_name) \
    { (type), OM_S_OBJECT_IDENTIFIER_STRING, \
      { { OMP_LENGTH(oid_name) , OMP_D_##oid_name } } }

/* Macro to mark the end of a public object.
 */
#define OM_NULL_DESCRIPTOR \
    { OM_NO_MORE_TYPES, OM_S_NO_MORE_SYNTAXES, \
      {0,OM_ELEMENTS_UNSPECIFIED} }

/* Macro to make class constants available within a compilation unit
 */
#define OM_IMPORT(class_name) \
    extern char OMP_D_##class_name [ ]; \
    extern OM_string class_name;

```

```
/* Macro to allocate memory for class constants within a compilation unit
*/
#define OM_EXPORT(class_name) \
    char OMP_D_##class_name[] = OMP_O_##class_name ; \
    OM_string class_name = \
        { OMP_LENGTH(class_name), OMP_D_##class_name };

/* Constant for the OM package
*/
#define OMP_O_OM_OM "\x56\x06\x01\x02\x04"

/* Constant for the Encoding class
*/
#define OMP_O_OM_C_ENCODING "\x56\x06\x01\x02\x04\x01"

/* Constant for the External class
*/
#define OMP_O_OM_C_EXTERNAL "\x56\x06\x01\x02\x04\x02"

/* Constant for the Object class
*/
#define OMP_O_OM_C_OBJECT "\x56\x06\x01\x02\x04\x03"

/* Constant for the BER Object Identifier
*/
#define OMP_O_OM_BER "\x51\x01"

/* Constant for the Canonical-BER Object Identifier
*/
#define OMP_O_OM_CANONICAL_BER "\x56\x06\x01\x02\x04\x04"
```

```
/* Return Code */
```

```
#define OM_SUCCESS ( (OM_return_code) 0 )
#define OM_ENCODING_INVALID ( (OM_return_code) 1 )
#define OM_FUNCTION_DECLINED ( (OM_return_code) 2 )
#define OM_FUNCTION_INTERRUPTED ( (OM_return_code) 3 )
#define OM_MEMORY_INSUFFICIENT ( (OM_return_code) 4 )
#define OM_NETWORK_ERROR ( (OM_return_code) 5 )
#define OM_NO_SUCH_CLASS ( (OM_return_code) 6 )
#define OM_NO_SUCH_EXCLUSION ( (OM_return_code) 7 )
#define OM_NO_SUCH_MODIFICATION ( (OM_return_code) 8 )
#define OM_NO_SUCH_OBJECT ( (OM_return_code) 9 )
#define OM_NO_SUCH_RULES ( (OM_return_code) 10 )
#define OM_NO_SUCH_SYNTAX ( (OM_return_code) 11 )
#define OM_NO_SUCH_TYPE ( (OM_return_code) 12 )
#define OM_NO_SUCH_WORKSPACE ( (OM_return_code) 13 )
#define OM_NOT_AN_ENCODING ( (OM_return_code) 14 )
#define OM_NOT_CONCRETE ( (OM_return_code) 15 )
#define OM_NOT_PRESENT ( (OM_return_code) 16 )
#define OM_NOT_PRIVATE ( (OM_return_code) 17 )
#define OM_NOT_THE_SERVICES ( (OM_return_code) 18 )
#define OM_PERMANENT_ERROR ( (OM_return_code) 19 )
#define OM_POINTER_INVALID ( (OM_return_code) 20 )
#define OM_SYSTEM_ERROR ( (OM_return_code) 21 )
#define OM_TEMPORARY_ERROR ( (OM_return_code) 22 )
#define OM_TOO_MANY_VALUES ( (OM_return_code) 23 )
#define OM_VALUES_NOT_ADJACENT ( (OM_return_code) 24 )
#define OM_WRONG_VALUE_LENGTH ( (OM_return_code) 25 )
#define OM_WRONG_VALUE_MAKEUP ( (OM_return_code) 26 )
#define OM_WRONG_VALUE_NUMBER ( (OM_return_code) 27 )
#define OM_WRONG_VALUE_POSITION ( (OM_return_code) 28 )
#define OM_WRONG_VALUE_SYNTAX ( (OM_return_code) 29 )
#define OM_WRONG_VALUE_TYPE ( (OM_return_code) 30 )
```

```
/* String (Elements component) */

#define OM_ELEMENTS_UNSPECIFIED      ((void *) 0)

/* Syntax */

#define OM_S_NO_MORE_SYNTAXES      ((OM_syntax) 0)
#define OM_S_BIT_STRING            ((OM_syntax) 3)
#define OM_S_BOOLEAN               ((OM_syntax) 1)
#define OM_S_ENCODING_STRING       ((OM_syntax) 8)
#define OM_S_ENUMERATION           ((OM_syntax) 10)
#define OM_S_GENERAL_STRING        ((OM_syntax) 27)
#define OM_S_GENERALISED_TIME_STRING ((OM_syntax) 24)
#define OM_S_GRAPHIC_STRING        ((OM_syntax) 25)
#define OM_S_IA5_STRING            ((OM_syntax) 22)
#define OM_S_INTEGER               ((OM_syntax) 2)
#define OM_S_NULL                  ((OM_syntax) 5)
#define OM_S_NUMERIC_STRING        ((OM_syntax) 18)
#define OM_S_OBJECT                ((OM_syntax) 127)
#define OM_S_OBJECT_DESCRIPTOR_STRING ((OM_syntax) 7)
#define OM_S_OBJECT_IDENTIFIER_STRING ((OM_syntax) 6)
#define OM_S_OCTET_STRING          ((OM_syntax) 4)
#define OM_S_PRINTABLE_STRING      ((OM_syntax) 19)
#define OM_S_TELETEX_STRING        ((OM_syntax) 20)
#define OM_S.UTC_TIME_STRING       ((OM_syntax) 23)
#define OM_S_VIDEOTEX_STRING       ((OM_syntax) 21)
#define OM_S_VISIBLE_STRING        ((OM_syntax) 26)

#define OM_S_LONG_STRING           ((OM_syntax) 0x8000)
#define OM_S_NO_VALUE              ((OM_syntax) 0x4000)
#define OM_S_LOCAL_STRING          ((OM_syntax) 0x2000)
#define OM_S_SERVICE_GENERATED     ((OM_syntax) 0x1000)
#define OM_S_PRIVATE               ((OM_syntax) 0x0800)
#define OM_S_SYNTAX                ((OM_syntax) 0x03FF)
```

```
/* Type */

#define OM_NO_MORE_TYPES          ( (OM_type) 0 )
#define OM_ARBITRARY_ENCODING    ( (OM_type) 1 )
#define OM_ASN1_ENCODING         ( (OM_type) 2 )
#define OM_CLASS                  ( (OM_type) 3 )
#define OM_DATA_VALUE_DESCRIPTOR ( (OM_type) 4 )
#define OM_DIRECT_REFERENCE       ( (OM_type) 5 )
#define OM_INDIRECT_REFERENCE     ( (OM_type) 6 )
#define OM_OBJECT_CLASS           ( (OM_type) 7 )
#define OM_OBJECT_ENCODING        ( (OM_type) 8 )
#define OM_OCTET_ALIGNED_ENCODING ( (OM_type) 9 )
#define OM_PRIVATE_OBJECT        ( (OM_type) 10 )
#define OM_RULES                  ( (OM_type) 11 )

/* Value Position */

#define OM_ALL_VALUES    ( (OM_value_position) 0xFFFFFFFF )

[/* WORKSPACE INTERFACE */
#include <xomi.h>]

/* END SERVICE INTERFACE */
```





# Workspace Interface

## 5.1 INTRODUCTION

This Chapter defines the workspace interface. The workspace interface defines types which specify the initial part of the representation of objects, and some associated data structures. This representation is mandatory, in order that interworking of services provided by different vendors can be achieved. The representation of objects beyond this is not specified (particularly the representation of attributes and values), and can be chosen by vendors to suit their individual needs.

The workspace interface is C-specific; there is no generic specification of it. Designers of additional programming language bindings to the OM specification will need to produce an appropriate solution in the alternative language.

The workspace interface also provides a macro definition, for each function in the service interface, which uses the defined data structures to call the implementation of the functions appropriate for the particular arguments. These are called the 'dispatcher' macros.

All implementations must provide an *<xom.h>* and *<xomi.h>* header containing the data types and macros and declarations defined in the C workspace interface, as defined in this specification, and this should be the default when application programs are compiled. Vendors may additionally provide alternative means of invoking the function definitions (by new definitions of the macros, or by function libraries). This might be used, for example, to provide additional error-checking capabilities. Such alternatives are vendor extensions, and the vendor may choose any appropriate method to select them.

## 5.2 REPRESENTATION OF OBJECTS

There are three types of objects, from a service implementation's point of view:

- **private objects (PRI)**. These are represented in an unspecified, private way in storage allocated by the service. The client is only able to access these objects by calling OM functions.
- **service-generated public objects (SPUB)**. These are represented as arrays of (*OM\_descriptor*), in storage allocated by the service. String values are also allocated in this way. The client may not modify any of this store. For example, it must not make assignments to any of the fields of the descriptors.
- **client-generated public objects (CPUB)**. These are represented as arrays of (*OM\_descriptor*), in storage allocated by the client. The client is responsible for management of this store and may freely modify it.

Implementations also need to discover whether a PRI or a SPUB belongs to their workspace, or to some other. These objects are allocated by a particular service implementation, in a workspace associated with that service. Note that:

- the internal representation (service view) and external representation (client view) of a CPUB are completely identical;
- the external representation of a CPUB and a SPUB are identical, and
- the internal representation of a SPUB and a PRI must provide additional information in order to be able to call the OM function implementation associated with each SPUB and PRI.

The third statement is achieved by basing the internal representation of all service-generated objects on a two-element descriptor array. We refer to these as the *-1st* and *0th* elements.

The external representation (i.e. the pointer returned to the client by *om\_create()*, *om\_get()*, etc.) points to the second, *0th*, descriptor. The client is not aware of the existence of the first, *-1st*, descriptor.

*Note that in the case of CPUBs and SPUBs there will usually be additional descriptors following the 0th, which are visible to the client.*

Then a service implementation can distinguish the type of object using the *type* and *syntax* components of the *0th* descriptor, as follows:

1. Inspect the *type* component. If it is *{OM\_PRIVATE\_OBJECT}*, the object is a PRI. Otherwise, it is a SPUB or CPUB.
2. Inspect the *{OM\_S\_SERVICE\_GENERATED}* bit. If this is set, the object is an SPUB. Otherwise, it is a CPUB.

If it is a PRI or a SPUB, the associated workspace pointer is stored in the *value.string.elements* component of the *-1st* descriptor, and the correct function implementation can be called using this.

Note that this means the storage for the workspace structure, and the function jump table, must remain allocated after the workspace has been shut down. In order to eventually reclaim this storage, implementations may use a reference count of the

number of service-generated descriptor arrays which have been allocated by *om\_get()* and not subsequently freed by *om\_delete()* .

Implementations need not, but may, allocate storage for the other components of the *-1st* descriptor (i.e. *type*, *syntax* and *value.string.length*) of PRIs and SPUBs. They may, but need not, allocate storage for the *value* component of the *0th* descriptor of a PRI. They must allocate the whole of the *0th* descriptor of a SPUB, since this forms part of the data returned to the client. Clients do not allocate the *-1st* descriptor of a CPUB, and the service must not refer to it.

Implementations may attach arbitrary private data in storage before or after the defined region of a PRI, and before the defined region of a SPUB. They may use this as they wish.

Note that a service-generated public object might reference private subobjects, the handles of which are no longer valid, for example, when the corresponding parent private object has been deleted. In order to determine whether a subobject in a service-generated public object is private, for example, when the service deletes a service-generated public object, the *{OM\_S\_PRIVATE}* bit in the public object's descriptor having the reference might be inspected.

Note also that if a Service-Generated Public Object has public subobjects (for example, due to an *OM\_get()* with no exclusions), the *-1st* descriptor will exist and the *OM\_S\_SERVICE\_GENERATED* bit will be set on all public subobjects.

### 5.3 TYPES AND MACROS

#### 5.3.1 Standard Internal Representation of an Object

The *OMP\_object\_header* and *OMP\_object* types provide the standard representation.

```
typedef OM_descriptor OMP_object_header[2];
typedef OMP_object_header *OMP_object;
```

The *{OM\_S\_SERVICE\_GENERATED}* bit in the Syntax component of a descriptor of Type *{OM\_CLASS}* determines whether a service allocated the storage for the descriptor array.

#### 5.3.2 Standard Internal Representation of a Workspace

```
typedef OM_return_code
(*OMP_copy) (
    OM_private_object    original,
    OM_workspace         workspace,
    OM_private_object    *copy
);

typedef OM_return_code
(*OMP_copy_value) (
    OM_private_object    source,
    OM_type              source_type,
    OM_value_position    source_value_position,
    OM_private_object    destination,
    OM_type              destination_type,
    OM_value_position    destination_value_position
);

typedef OM_return_code
(*OMP_create) (
    OM_object_identifier class,
    OM_boolean           initialise,
    OM_workspace         workspace,
    OM_private_object    *object
);

typedef OM_return_code
(*OMP_decode) (
    OM_private_object    encoding,
    OM_private_object    *original
);

typedef OM_return_code
(*OMP_delete) (
    OM_object            subject
);
```

```

typedef OM_return_code
(*OMP_encode) (
    OM_private_object    original,
    OM_object_identifier rules,
    OM_private_object    *encoding
);

typedef OM_return_code
(*OMP_get) (
    OM_private_object    original,
    OM_exclusions        exclusions,
    OM_type_list         included_types,
    OM_boolean           local_strings,
    OM_value_position    initial_value,
    OM_value_position    limiting_value,
    OM_public_object     *copy,
    OM_value_position    *total_number
);

typedef OM_return_code
(*OMP_instance) (
    OM_object            subject,
    OM_object_identifier class,
    OM_boolean           *instance
);

typedef OM_return_code
(*OMP_put) (
    OM_private_object    destination,
    OM_modification      modification,
    OM_object            source,
    OM_type_list         included_types,
    OM_value_position    initial_value,
    OM_value_position    limiting_value
);

typedef OM_return_code
(*OMP_read) (
    OM_private_object    subject,
    OM_type              type,
    OM_value_position    value_position,
    OM_boolean           local_string,
    OM_string_length     *string_offset,
    OM_string            *elements
);

```

```

typedef OM_return_code
(*OMP_remove) (
    OM_private_object  subject,
    OM_type            type,
    OM_value_position  initial_value,
    OM_value_position  limiting_value
);

typedef OM_return_code
(*OMP_write) (
    OM_private_object  subject,
    OM_type            type,
    OM_value_position  value_position,
    OM_syntax          syntax,
    OM_string_length   *String_offset,
    OM_string          elements
);

typedef struct OMP_functions_body {
    OM_uint32          function_number;
    OMP_copy           omp_copy;
    OMP_copy_value     omp_copy_value;
    OMP_create         omp_create;
    OMP_decode         omp_decode;
    OMP_delete         omp_delete;
    OMP_encode         omp_encode;
    OMP_get            omp_get;
    OMP_instance       omp_instance;
    OMP_put            omp_put;
    OMP_read           omp_read;
    OMP_remove         omp_remove;
    OMP_write          omp_write;
} OMP_functions;

typedef struct OMP_workspace_body {
    struct OMP_functions_body  *functions;
} *OMP_workspace;

```

**Description**

A data value of this data type is the designator or handle for a workspace, refined for the workspace interface.

The first component of a data value of type *OMP\_functions\_body*, **Function Number**, is the number of the other components, which are implementations of the workspace interface functions, those pertaining to (private) objects in the workspace.

*Note that the purpose of the **Function Number** component is to enable functions to be backward-compatibly added to the workspace interface to create future versions of it.*

*Note that because the interface calls for workspaces to provide the storage for all data values of this data type, the above information may be accompanied (e.g., followed in memory) by other pieces of information whose number, types and purposes are outside the scope of this document and workspace-specific. In the context of a particular operating system, this information may be subject to system-wide agreements designed, for example, to facilitate storage management.*

### 5.3.3 Useful Macros

The following macro converts (a pointer to) the internal representation of an object to (a pointer to) the external representation.

```
#define OMP_EXTERNAL(internal) \
    ((OM_object)((OM_descriptor*)(internal) + 1))
```

The following macro converts (a pointer to) the external representation of an object to (a pointer to) the internal representation.

```
#define OMP_INTERNAL(external) ((OM_descriptor*)(external) - 1)
```

The following macro extracts the type component of a descriptor, given the pointer to it.

```
#define OMP_TYPE(desc) (((OM_descriptor*)(desc))->type)
```

The following macro extracts the workspace of an object, given the external pointer to it. The effect of applying it to a client-generated public object is undefined.

```
#define OMP_WORKSPACE(external) \
    ((OMP_workspace)(OMP_INTERNAL(external)->value.string.elements))
```

The following macro extracts the function jump-table associated with an object, given the external pointer to it. The effect of applying it to a client-generated public object is undefined.

```
#define OMP_FUNCTIONS(external) (OMP_WORKSPACE(external)->functions)
```



## 5.4 DISPATCHER MACROS

The dispatcher macros provide the interface between application programs and implementations of the OM service.

```
#define om_copy(ORIGINAL,WORKSPACE,COPY) \
  (((OMP_workspace)(WORKSPACE))->functions->omp_copy( \
    (ORIGINAL),(WORKSPACE),(COPY)))

#define om_copy_value(SOURCE, SOURCE_TYPE, SOURCE_POSITION, \
  DEST, DEST_TYPE, DEST_POSITION) \
  (OMP_FUNCTIONS(DEST)->omp_copy_value( \
    (SOURCE), (SOURCE_TYPE), (SOURCE_POSITION), \
    (DEST), (DEST_TYPE), (DEST_POSITION)))

#define om_create(CLASS,INITIALISE,WORKSPACE,OBJECT) \
  (((OMP_workspace)(WORKSPACE))->functions->omp_create( \
    (CLASS),(INITIALISE),(WORKSPACE),(OBJECT)))

#define om_decode(ENCODING,ORIGINAL) \
  (OMP_FUNCTIONS(ENCODING)->omp_decode( (ENCODING),(ORIGINAL)))

#define om_delete(SUBJECT) \
  (((SUBJECT)->syntax & OM_S_SERVICE_GENERATED) ? \
  OMP_FUNCTIONS(SUBJECT)->omp_delete( (SUBJECT)) : \
  OM_NOT_THE_SERVICES)

#define om_encode(ORIGINAL,RULES,ENCODING) \
  (OMP_FUNCTIONS(ORIGINAL)->omp_encode( (ORIGINAL),(RULES),(ENCODING)))

#define om_get(ORIGINAL,EXCLUSIONS,TYPES,LOCAL_STRINGS, \
  INITIAL,LIMIT,COPY,TOTAL_NUMBER) \
  (OMP_FUNCTIONS(ORIGINAL)->omp_get( \
    (ORIGINAL),(EXCLUSIONS),(TYPES),(LOCAL_STRINGS), \
    (INITIAL),(LIMIT),(COPY),(TOTAL_NUMBER)))

#define om_instance(SUBJECT,CLASS,INSTANCE) \
  (((SUBJECT)->syntax & OM_S_SERVICE_GENERATED) ? \
  OMP_FUNCTIONS(SUBJECT)->omp_instance( (SUBJECT),(CLASS),(INSTANCE)) : \
  OM_NOT_THE_SERVICES)

#define om_put(DESTINATION,MODIFICATION,SOURCE,TYPES,INITIAL,LIMIT) \
  (OMP_FUNCTIONS(DESTINATION)->omp_put( \
    (DESTINATION),(MODIFICATION),(SOURCE),(TYPES),(INITIAL),(LIMIT)))

#define om_read(SUBJECT,TYPE,VALUE_POS,LOCAL_STRING, \
  STRING_OFFSET,ELEMENTS) \
  (OMP_FUNCTIONS(SUBJECT)->omp_read( \
    (SUBJECT),(TYPE),(VALUE_POS),(LOCAL_STRING), \
    (STRING_OFFSET),(ELEMENTS)))
```

```
#define om_remove(SUBJECT,TYPE,INITIAL,LIMIT)      \  
    (OMP_FUNCTIONS(SUBJECT)->omp_remove(         \  
        (SUBJECT),(TYPE),(INITIAL),(LIMIT)))  
  
#define om_write(SUBJECT,TYPE,VALUE_POS,SYNTAX,STRING_OFFSET,ELEMENTS)  \  
    (OMP_FUNCTIONS(SUBJECT)->omp_write(  \  
        (SUBJECT),(TYPE),(VALUE_POS),  \  
        (SYNTAX),(STRING_OFFSET),(ELEMENTS)))
```

# *Object Management Package*

## 6.1 INTRODUCTION

This Chapter defines the OM package. The object identifier, referred to symbolically as *om*, that is assigned to the package, as defined by this edition of this document, is that specified in ASN.1 as *{joint-iso-ccitt mhs-motis(6) group(6) white(1) api(2) om(4)}*.

## 6.2 CLASS HIERARCHY

This section depicts the hierarchical organisation of the OM classes. Subclassification is indicated by indentation. The names of abstract classes are in italics. Thus, for example, Encoding is an immediate subclass of Object, an abstract class. The names of classes to which the Encode function applies are in **bold**. The Create function applies to all concrete classes.

*Object*

- Encoding
- **External**

### 6.3 CLASS DEFINITIONS

This section defines the OM classes.

#### 6.3.1 Encoding

An instance of class **Encoding** is an object represented in a form suitable for conveyance between workspaces, transport via a network, or storage in a file. An encoding also may be a suitable way to present to an intermediate service provider (e.g. a directory or message transfer system) an object it does not recognise.

This class has the attributes of its superclass (*Object*) and the specific attributes listed in Table 11 below.

Attribute	Value Syntax	Value Length	Value Number	Value Initially
Object Class	String (Object Identifier)	-	1	-
Object Encoding	String <sup>1</sup>	-	1	-
Rules	String (Object Identifier)	-	1	ber

<sup>1</sup>If the Rules attribute is **ber** or **canonical-ber**, the syntax of the present attribute shall be String (Encoding).

**Table 11: Attributes Specific to Encoding**

**Object Class** Identifies the class of the object that the Object Encoding attribute encodes. The class shall be concrete.

**Object Encoding** The encoding itself.

**Rules** Identifies the set of rules that were followed to produce the Object Encoding attribute. Among the defined values of this attribute are those referred to symbolically as follows:

- **ber**. Specified in ASN.1 as *{joint-iso-ccitt asn1(1) basic-encoding(1)}*, this value denotes the BER (see Clause 25.2 of Recommendation X.208, **Referenced Documents**).
- **canonical-ber**. Specified in ASN.1 as *{joint-iso-ccitt mhs-motis(6) group(6) white(1) api(2) om(4) canonical-ber(4)}*, this value denotes the canonical BER (see Clause 8.7 of Recommendation X.509, **Referenced Documents**).

*Note that an instance of this class may not appear, in general, as a value whose syntax is Object (C), if C is not Encoding, even if the class of the object encoded is C.*

#### 6.3.2 External

An instance of class **External** is a data value, not necessarily describable using ASN.1, and one or more information items that describe the data value and identify its data type. This class corresponds to ASN.1's External type, and thus the class and the attributes specific to it are more fully described, indirectly, in Clause 34 of Recommendation X.208 (see **Referenced Documents**).

This class has the attributes of its superclass (*Object*) and the specific attributes listed in Table 12.

Attribute	Value Syntax	Value Length	Value Number	Value Initially
Arbitrary Encoding	String (Bit)	-	0-1 <sup>1</sup>	-
ASN1 Encoding	String (Encoding)	-	0-1 <sup>1</sup>	-
Data Value Descriptor	String (Object Descriptor)	-	0-1	-
Direct Reference	String (Object Identifier)	-	0-1	-
Indirect Reference	Integer	-	0-1	-
Octet Aligned Encoding	String (Octet)	-	0-1 <sup>1</sup>	-

<sup>1</sup>Exactly one of these three attributes shall be present.

**Table 12: Attributes Specific to External**

### Arbitrary Encoding

A representation of the data value as a bit string. This attribute is described more fully in Clause 34 of Recommendation X.208 (see **Referenced Documents**).

### ASN1 Encoding

The data value. This attribute may be present only if the data type is an ASN.1 type. This attribute is described more fully in Clause 34 of Recommendation X.208 (see **Referenced Documents**).

If this attribute value's syntax is an Object syntax, the data value's implied representation is that produced by the Encode function when its Object argument is the attribute value and its Rules argument is **ber**. Thus the object's class shall be one to which the Encode function applies.

### Data Value Descriptor

A description of the data value. This attribute is described more fully in Clause 34 of Recommendation X.208 (see **Referenced Documents**).

### Direct Reference

A direct reference to the data type. This attribute is described more fully in Clause 34 of Recommendation X.208 (see **Referenced Documents**).

### Indirect Reference

An indirect reference to the data type. This attribute is described more fully in Clause 34 of Recommendation X.208 (see **Referenced Documents**).

### Octet Aligned Encoding

A representation of the data value as an octet string. This attribute is described more fully in Clause 34 of Recommendation X.208 (see **Referenced Documents**).

### 6.3.3 Object

The class *Object* represents information objects of any variety. This abstract class is distinguished by the fact that it has no superclass and that all other classes are its subclasses.

The attributes specific to this class are listed in Table 13 below.

Attribute	Value Syntax	Value Length	Value Number	Value Initially
Class	String (Object Identifier)	-	1	-

**Table 13: Attributes Specific to Object**

**Class** Identifies the object's class.

This appendix is not an integral part of the document. It repeats the definitions of the technical terms defined in the body of the document.

<b>Abstract</b>	Said of a class, instances of which are forbidden.
<b>Accessible</b>	Said of an object for which the client possesses a valid designator or handle.
<b>Attribute</b>	A component of an object, comprising an integer denoting the attribute's type and an ordered sequence of one or more attribute values, each accompanied by an integer denoting the value's syntax.
<b>Bit String</b>	A string comprising bits.
<b>C Interface</b>	The interface, defined at a level that depends upon the variant of C standardised by ANSI.
<b>Character String</b>	A string comprising characters.
<b>Class</b>	A category into which objects are placed on the basis of both their purpose and their internal structure.
<b>Client</b>	Software that uses the interface.
<b>Concrete</b>	Said of a class, instances of which are permitted.
<b>Descriptor</b>	The means by which the client and service exchange an attribute value and the integers that denote its representation, type and syntax.
<b>Dispatcher</b>	The software that implements the service interface functions using workspace interface functions.
<b>Element</b>	Any of the bits of a bit string, the octets of an octet string, or the octets by means of which the characters of a character string are represented.
<b>Generic Interface</b>	The interface, defined at a level that is independent of any particular programming language
<b>Immediate Subclass</b>	A subclass, of a class C, having no superclasses that are themselves subclasses of C.
<b>Immediate Subobject</b>	One object that is a value of an attribute of another.
<b>Immediate Superclass</b>	The superclass, of a class C, having no subclasses that are themselves superclasses of C.
<b>Immediate Superobject</b>	One object that contains another among its attribute values.

<b>Inaccessible</b>	Said of an object for which the client does not possess a valid designator or handle.
<b>Instance</b>	An object in the category represented by a class.
<b>Interface</b>	The interface this document defines.
<b>Intermediate Data Type.</b>	Any of the basic data types in terms of which the other, substantive data types of the interface are defined.
<b>Intermediate Macro</b>	In the C interface, any of the basic macros in terms of which the other, substantive macros used to realise the dispatcher are defined.
<b>Length</b>	The number of elements in a string.
<b>Minimally Consistent</b>	Said of an object that satisfies various conditions set forth in the definition of its class.
<b>Object Management</b>	The creation, examination, modification and deletion of potentially complex information objects.
<b>Object</b>	Any of the complex information objects created, examined, modified or destroyed by means of the interface.
<b>Octet String</b>	A string comprising octets.
<b>OSI Object Management Application Program Interface</b>	The interface this document defines.
<b>Package</b>	A group of related classes.
<b>Package Closure</b>	The set of classes that need to be supported in order to be able to create all possible instances of all classes defined in the package.
<b>Position (within a string)</b>	The ordinal position of one element of a string relative to another.
<b>Position (within an attribute)</b>	The ordinal position of one value relative to another.
<b>Primary Representation</b>	The form in which the service supplies an attribute value to the client.
<b>Private Object</b>	An object that is represented in an unspecified fashion.
<b>Public Object</b>	An object that is represented by a data structure whose format is part of the service's specification.
<b>Secondary Representation</b>	A second form, an alternative to the primary representation, in which the client may supply an attribute value to the service.
<b>Segment</b>	Zero or more contiguous elements of a string.



<b>Service Interface</b>	The interface as realised, for the client's benefit, by the service as a whole.
<b>Service</b>	Software that implements the interface.
<b>Specific</b>	Said, with respect to a class, of the attribute types that may appear in an instance of the class but not in an instance of its superclasses.
<b>String</b>	An ordered sequence of zero or more bits, octets or characters, accompanied by the string's length.
<b>Subclass</b>	One of the classes, designated as such, whose attribute types are a superset of those of another class.
<b>Subobject</b>	An immediate subobject of an object or of one of its subobjects.
<b>Superclass</b>	One of the classes, designated as such, whose attribute types are a subset of those of another class.
<b>Superobject</b>	An object's immediate superobject, or one of its superobjects.
<b>Syntax template</b>	A lexical construct containing an asterisk from which several attribute syntaxes can be derived by substituting text for the asterisk.
<b>Syntax</b>	A category into which an attribute value is placed on the basis of its form.
<b>Type</b>	A category into which attribute values are placed on the basis of their purpose.
<b>Value</b>	An arbitrarily complex information item that can be viewed as a characteristic or property of an object.
<b>Workspace Interface</b>	The interface as realised, for the dispatcher's benefit, by each workspace individually.
<b>Workspace</b>	A repository for instances of classes in the closures of one or more packages associated with the workspace.



# *Index*

- 1st descriptor: 68
- 0th descriptor: 68
- 0th element: 68
- 1st element: 68
- ANSI: 5
- any: 10
- API: 1, 5
- Application Program Interface: 1
- architecture: 7, 10
- ASN.1: 2, 5, 15-16, 18, 77
- attribute: 8
- attribute values: 15
- Basic Encoding Rules (BER): 1
- BER: 5, 16
- boolean: 23
- C interface: 3, 15, 22, 58, 67
- CCITT: 5
- class: 9
- class:
  - abstract: 9
  - concrete: 9
  - constraints: 10
  - definitions: 10
- class closure: 11
- client: 2, 7, 58
- client-generated public object: 68
- closure: 11
- copy: 36
- copy value: 37
- create: 39
- data types: 22
- decode: 40
- delete: 41
- descriptor: 8, 23
- descriptors: 3
- dispatcher macros: 67, 75
- encode: 42
- encoding: 78
- enumeration: 25
- exclusions: 25
- external: 78
- functions:
  - copy: 36
  - copy value: 37
  - create: 39
  - decode: 40
  - delete: 41
  - encode: 42
  - get: 43
  - instance: 47
  - put: 48
  - read: 50
  - remove: 52
  - return codes: 55
  - write: 53
- functions summary: 34
- generic interface: 3, 55
- get: 43
- handle: 13, 33
- header file: 58
- IA5: 5
- instance: 47
- integer: 25
- intermediate data types: 22
- ISO: 5
- macros: 74
- object: 13, 68, 80
  - handle: 26
  - standard internal representation: 70
- object identifier: 9, 26
- OM: 5
- OM API: 1
- OM class:
  - encoding: 78
  - external: 78
  - object: 80
- OM classes: 77
- OM package: 77
- OSI: 5
- OSI Object Management: 1
- package: 9, 11, 77
- package closure: 11
- private object: 13, 28, 68
  - accessible: 13
  - inaccessible: 13
- public object: 13, 28
- put: 48
- read: 50

- remove: 52
- return code: 29
- return codes: 55
- segment: 17
- service: 2
- service interface: 21
  - copy: 36
  - copy value: 37
  - create: 39
  - data types: 22
  - declarations: 58
  - decode: 40
  - delete: 41
  - encode: 42
  - functions: 34
  - get: 43
  - instance: 47
  - put: 48
  - read: 50
  - remove: 52
  - return codes: 55
  - write: 53
- service-generated public object: 68
- storage: 13
- string: 16, 29
  - data type: 16
  - segment: 17
  - value length: 16
- syntax: 8, 15, 30, 68
  - template: 15
- template: 15
- type: 8, 15, 31, 68
- type list: 31
- UTC: 5
- value: 8, 32
- value length: 32
- value position: 33
- workspace: 1, 3, 12-13, 33
  - designator: 33
  - handle: 33
  - standard internal representation: 70
- workspace interface: 67
- write: 53
- xom.h: 58, 67
- xomi.h: 58, 67