X/Open CAE Specification

OSI-Abstract-Data Manipulation API (XOM), Issue 2

X/Open Company Ltd.

© February 1994, X/Open Company Limited

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 the copyright owners.

X/Open CAE Specification

OSI-Abstract-Data Manipulation API (XOM), Issue 2

ISBN: 1-85912-008-3
X/Open Document Number: C315

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

Contents

List of Figures

List of Tables

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 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). It defines the application programming interface (API) to management of Open Systems Interconnection (OSI) objects. This interface is needed by other APIs specific to particular OSI services. Currently, these include the X/Open APIs to Directory Services (XDS), Electronic Mail (X.400), and Systems Management Protocols (XMP).

This *Issue 2* of the XOM CAE Specification includes revisions to align with the IEEE OSI Abstract Data Manipulation group of standards, that themselves are based on the previous version of this X/Open specification.

All new implementation work by API providers should be based on this Issue 2. The previous specification will be retained by X/Open for only so long as branding is available for products based on it.

XOM is one of several specifications that X/Open originally developed in collaboration with the X.400 API Association. The other documents are XDS, X.400, XMS and XEDI API specifications, and a Guide to Selected X.400 and Directory Services APIs.

The XDS and X.400 specifications have similarly served as bases for corresponding IEEE standards. X/Open has now also revised the XDS and X.400 specifications into *Issue 2* publications, to align them with the corresponding IEEE Standards.

Structure

This document is organised as follows:

- Chapter 1 identifies the scope and purpose of this API, explains the C naming conventions, and conformance criteria for conformant implementations. It also lists commonly used abbreviations used.
- Chapter 2 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 and Chapter 5 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.

- Chapter 3 defines the permitted syntaxes of attribute values.
- Chapter 4 defines the service interface, in terms of 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. It also summarises the declarations that define the C service interface.
- Chapter 5 defines the workspace interface, which defines types which specify the initial part of the representation of objects, and some associated data structures.
- Chapter 6 defines the OM package.
- Appendix A identifies the known substantive differences between this X/Open **OSI-Abstract-Data Manipulation (XOM)** API specification and the corresponding **IEEE Object Management Standard**.

A glossary and index are provided.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members, and language-independent names.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  — command operands, command option-arguments or variable names, for example, substitutable argument prototypes
  — environment variables, which are also shown in capitals
  — utility names
  — external variables, such as *errno*
  — functions; these are shown as follows: *name*( ). Names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Roman font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
- The notation [EABCD] is used to identify a return value ABCD, including if this is an an error value.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font. Brackets shown in this font, `[ ]`, are part of the syntax and do *not* indicate optional items.
- Further details on the C naming conventions used in this specificaction are given in Section 1.4 on page 3.

Trade

X/Open^{TM} and the ‘‘X’’ device are trade marks of X/Open Company Ltd.

Referenced Documents

ANSI-C
: Information Processing: Programming Language C, ISO Draft International Standard DIS9899 (also known as *ANSI C*, American National Standard X3.159-1989).

ASN.1
: ISO 8824: 1990, CCITT X.208: 1988. Information Technology - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1),

BER
: ISO / IEC 8825:1990 (ITU-T Recommendation X.209 (1988)), Information Technology — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

IEEE 1224-1993
: IEEE 1224-1993: IEEE Standard for Information Technology - Open Systems Interconnection (OSI) Abstract Data Manipulation - Application Programming Interface (API) [Language Independent], ISBN 1-55937-301-6.

IEEE 1327-1993
: IEEE 1327-1003: IEEE Standard for Information Technology - Open Systems Interconnection (OSI) Abstract Data Manipulation C Language Interfaces - Binding for Application Programming Interface (API), ISBN 1-55937-311-3.

X.400
: X / Open CAE Specification, February 1994, API to Electronic Mail (X.400), Issue 2, (ISBN: 1-85912-009-1, X / Open document C316).

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

XDS
: X / Open CAE Specification, February 1994, API to Directory Services (XDS), Issue 2, (ISBN: 1-85912-007-5, X / Open document C317).

XPG4
: X / Open Systems and Branded Products: XPG4, July 1992 (ISBN: 1-872630-52-9, X924).

Chapter 1

Introduction

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[1]. It presents to programmers a uniform model, or architecture, of information based upon the concept of groups, or classes, of similar information objects. The OM API provides facilities to manipulate both small objects and those too large to be held in main memory.

This API is designed to work with groups of ASN.1 objects that are called packages. These packages are defined by the application specific APIs that use this API. The packages contain all the ASN.1 objects necessary to accomplish a specific task. Thus, the API does not work with any arbitrary ASN.1 objects, only those defined in packages.

The information objects to which OM applies are those that arise in OSI, that is, 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, for example, its Basic Encoding Rules (BER).

The OM API is designed to be implemented by one or more manufacturers working independently. As illustrated in the figure 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 on page 12).

1. These objects, which are described in Chapter 2, differ from the directory objects defined in ISO 9594.

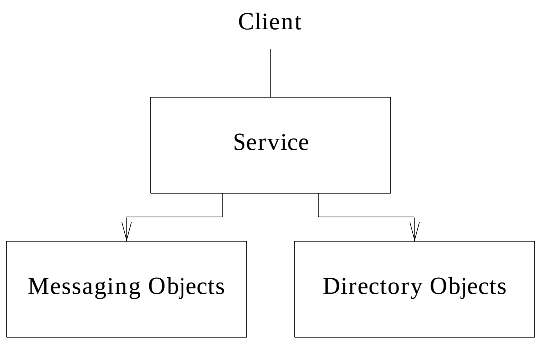

Figure 1-1 Conceptual Model of Object Management

Throughout this document, the term 'interface' denotes the OM API, the term 'service' denotes software that supplies (that is, 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, for example, 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. 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: 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 on page 12).

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 the following table. 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)	*none*	Lower
Function	om_	Lower
Function argument	*none*	Lower
Function result	*none*	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 1-1 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:

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

  Note: 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.

- **Workspaces.**
  The service shall comprise one or more workspaces (see Chapter 4), each of which shall support the OM package (see Chapter 5).

- **Aspects.**
  The service shall implement all defined aspects of the interface, subject to the options summarised in Section 1.5.
- **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
CPUB	Client-generated Public Object
IA5	International Alphabet No. 5
ISO	International Organisation for Standardisation
OM	Object Management
OSI	Open Systems Interconnection
PRI	Private Object
SPUB	Service-generated Public Object
UTC	Universal Coordinated Time
XDS	(X/Open) Directory Services
X.400	X.400-Based Electronic Messaging

Chapter 2

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 and Chapter 5, 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 on page 21). 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, that is, 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, that is, by means of programming language constructs.

Note: 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 2-1 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** (for example, 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` (`Integer`) is a category into which a value is placed on the basis of its form. A **type** (for example, **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.

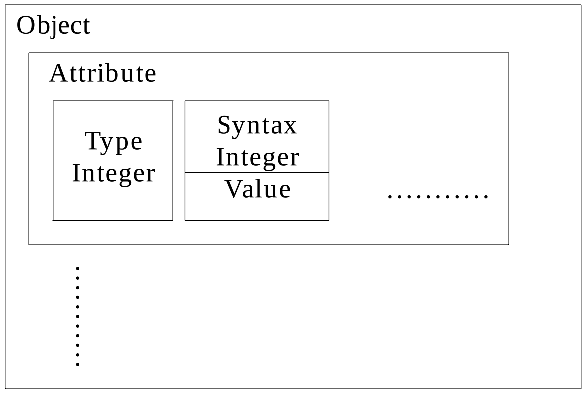

Figure 2-1 Structure of an Object

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 on page 11).

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 on page 11) 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 (for example, a message) is said to be an *instance* of its class (for example, *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.

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 on page 11). 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 on page 11) 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 on page 78). The class hierarchy is fixed by the classes' definitions (see Section 2.5 on page 10); 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 on page 10); 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 (that is, 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: 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: 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 on page 12) 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 on page 11), the definition of a class has the following elements:

- the class' name, which denotes the class' object identifier
- identification of the class' immediate superclass
- the definitions of the attribute types specific to the class
- an indication of whether the class is abstract or concrete.

The types specific to a class are themselves defined by means of a table (see Chapter 6 on page 75). The table gives:

- under the heading **Attribute**, the name of each attribute
- under the heading **Value Syntax**, the syntax or syntaxes of each of its values
- under the heading **Value Length**, any constraints upon the number of bits, octets or characters in each value that is a string
- under the heading **Value Number**, any constraints upon the number of values
- 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:

- no more than a fixed number of values
- either zero or one value, that value thus being optional
- 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:

- the type of each of its attributes is specific to the object's class or one of its superclasses
- the number of values of each attribute is no greater than the class permits
- the syntax of each value is among those the class permits
- 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
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.

Notes:

1. 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.
2. 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: the handles to these private subobjects stay valid. 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, but the handles to the private sub-objects are invalidated. 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.

Chapter 3

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 on page 21).

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: *$identifier_1$ ($identifier_2$)*.

3.3 Syntaxes

A variety of syntaxes are defined. Most are functionally equivalent to ASN.1 types, as documented in Section 3.6 through to Section 3.9.

The following syntaxes are defined:

Boolean
A value of this syntax is a Boolean, that is, 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.

Real
A value of this syntax is a real number, that is, it is composed of a (positive or negative) mantissa and an integer exponent.

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), 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.

Note: The value length of an attribute value that is a string may not be equal to the number of octets required to represent the value, as for some character encodings the representation of a single character may require several octets.

The syntaxes that form the String group are identified in Table 3-1 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[1]	General
	Object Identifier[2]	Generalised Time
	Octet	Graphic
		IA5
		Numeric
		Object Descriptor
		Printable
		Teletex
		UTC Time
		Universal
		Unrestricted[3]
		Videotex
		Visible

[1] The encoding method used to generate the string contents is determined by the context in which this syntax is used. For example, when used in the context of an object of class Encoding, the Rules attribute of the object defines which encoding method was used to generate the string (currently limited to either ber or canonical-ber).

[2]The octets are those the BER permit for the contents octets of the encoding of a value of ASN.1's Object Identifier type.

[3]Values of this syntax are represented in their BER encoded form.

Table 3-1 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 on page 29. 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}).

Notes:

1. An "octet" is defined as an ordered sequence of eight bits. Octets can be stored in larger objects if appropriate to a particular architecture.
2. 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 3-2 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	Real

Table 3-2 Syntax for ASN.1's Simple Types

3.7 Relationship to ASN.1 Useful Types

As shown in Table 3-3 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 3-3 Syntaxes for ASN.1's Useful Types

3.8 Relationship to ASN.1 Character String Types

As shown in Table 3-4 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)
Universal String	String(Universal)
Unrestricted String	String(Unrestricted)
Videotex String	String (Videotex)
Visible String	String (Visible)

Table 3-4 Syntaxes for ASN.1's Character String Types

3.9 Relationship to ASN.1 Type Constructors

As shown in Table 3-5 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	*any*[1]
Choice	Object
Enumerated	Enumeration
Selection	*none*[2]
Sequence	Object
Sequence Of	Object
Set	Object
Set Of	Object
Tagged	*none*[3]

[1] This type constructor denotes that any one of the other valid OM syntaxes are permitted as a value for the OM attribute, within the restrictions laid out for the interpretation of this syntax for a particular package.

[2]This type constructor, a purely specification-time phenomenon, has no corresponding syntax.

[3]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 3-5 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** or **Set**
  An attribute type may be defined for each sequence or set element.
- **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.

If an element of an ASN.1 construct is *optional*, then it is recommended that its corresponding OM attribute in the service's class definitions should also be defined as optional (that is, Value Number 0-1, or 0-or-more). This leads to consistency in the manner in which classes are defined.

Note: Some packages, which were defined using a previous version of this specification, may use a different convention, where an optional enumeration *sequence* element is represented by defining an additional enumeration member, **not-present**.

Chapter 4

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 the following table 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.
Real	type definition for Real data value.
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 4-1 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.
Double	The positive and negative floating point numbers representable in 64 bits.

The **OM_sint** and **OM_uint** data types are defined above by the ranges of integers they must accommodate. Implementations can define them by the range of integers the host machine's word size permits. The following examples will work on most machines.

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

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, that is, it is 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: 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);
```

Alternatively, the macros OM_SYNTAX(d) and OM_SYNTAX_ASSIGN(d,s) may be used for this purpose:

```
my_syntax = OM_SYNTAX(my_public_object[3]) ;
            OM_SYNTAX_ASSIGN(my_public_object[4], my_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).

The macro OM_IS_LONG_STRING(d) may be used to test this flag.

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).

The macro OM_HAS_VALUE(d) may be used to test this flag.

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).

The macro OM_IS_LOCAL_STRING(d) may be used to test this flag, and OM_SET_LONG_STRING(d) to set it.

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 4).

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

The macro OM_IS_SERVICE_GENERATED(d) may be used to test this flag.

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: 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).

The macro OM_IS_PRIVATE(d) may be used to test this flag.

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
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
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 Example 4-0 on page 58).

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 (for example, 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 xxx 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_xxx),
    { 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: 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 4 and Chapter 5. 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 and Value component are wholly unspecified.

4.2.11 Real

NAME

Real - type definition for a Real data value

C DECLARATION

```
typedef OM_double OM_real;
```

DESCRIPTION

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

4.2.12 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 on page 34.

This document employs integers in the (narrower) interval [0, 2^{15}) to denote the return codes they define.

4.2.13 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 (that is, 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.

	2nd octet								3rd octet		
position in bit string:	0	1	2	3	4	5	6	7	8	9	. . .
bit position in octet:	7	6	5	4	3	2	1	0	7	6	. . .
	↑							↑			
	most significant bit							least significant bit			

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. *Universal* and *Unrestricted* strings can contain null octets. Only the length-specified form shall be used to represent strings of these types. When either form is used, a null character (that is, an instance of the character whose encoding is zero) follows.

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.

4.2.14 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, 2^9) that denotes an individual syntax, or a set of syntaxes taken together.

The data value is chosen from among the following:

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

This document employs 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.

4.2.15 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.

This document employs integers in the (narrower) interval [0, 2^{15}) to denote the types they define.

4.2.16 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 unordered 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.

4.2.17 Value

NAME

Value - type definition for representing any data value

C DECLARATION

```
typedef union OM_value_union {
    OM_string           string;
    OM_boolean          boolean;
    OM_enumeration      enumeration;
    OM_integer          integer;
    OM_real             real;
} 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.

Data values of this type are used for representing all types of values, including objects. String, boolean, enumeration and integer values are accessed directly, by dereferencing the appropriate member of the OM_value union. Object values, however, are accessed indirectly using the following macros:

#define OM_OBJ_VALUE(v) ((OM_object)((v).string.elements))
Converts a value v of type OM_value into a value of type OM_object.

OM_OBJ_VALUE_ASSIGN(v,o) (((v).string.elements = o)
Assigns to a value v of type OM_value the object o of type OM_object.

Note: In order to provide backwards compatibility with applications that access an object identifier using the syntax `value.object.object`, an implementation may supply the following alternative definition.

```
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;
```

Note: This is designed to allow static initialisation of public objects. While it is realised that Standard C does not guarantee that a union of structures will have anything but their first elements aligned, these constructs will work on most architectures and compilers.

DESCRIPTION

A data value of this data type is an attribute value.

Note: The value length of an attribute value that is a string may not be equal to the number of octets required to represent the value, as for some character encodings the representation of a single character may require several octets.

Its components are wholly unspecified if the value's type is **no-more-types**, or if the value's syntax is ***no-value***. It has (exactly) one of the following components otherwise:

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

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

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

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

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

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

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.

4.2.18 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: This data type is not used in the definition of the interface. It is provided for use by client programmers in defining attribute constraints.

4.2.19 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.

4.2.20 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: The nature of the handle is of no concern to the client. However, the nature of the handle is of concern to service implementors and is therefore further specified in Chapter 4.

4.3 Functions

This section defines, and the following table 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 on page 58, 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 4-2 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:

Copy
This function creates an independent copy of an existing private object, and all of its subobjects. The copy is placed in the workspace specified by the client.

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.

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.

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.

Delete
This function deletes a service-generated public object, or makes a private object inaccessible.

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.

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).

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.

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.

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.

Remove

This function removes and discards particular values of an attribute of a private object. The attribute itself is removed if no values remain.

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, that is, 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 (
    const OM_private_object      original,
    const 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 (
    const OM_private_object       source,
    const OM_type                 source_type,
    const OM_value_position       source_value_position,
    OM_private_object             destination,
    OM_type                       destination_type,
    OM_value_position             destination_value_position
);
```

DESCRIPTION

This operation 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. If the source value does not have the syntax `String(*)`, the error code wrong-value-syntax shall be returned.

If the value of the destination attribute exists, the resulting value shall have the syntax of the replaced value. If the source value does not have the same syntax as the replaced value, the error code wrong-value-syntax shall be returned.

If the value of the destination attribute does not exist, the new value shall have the syntax of the source value. If the syntax of the source value is not compatible with the description of the destination attribute, the error code wrong-value-syntax shall be returned.

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 (
    const OM_object_identifier       class,
    const OM_boolean                 initialize,
    const 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 and shall be among those that are permissible to be created for its associated package.

Initialize (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: 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 (
    const 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-declined, 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 prohibited 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. The handles to the subobjects will be invalid. There is no effect on any private subobjects. The user should not use *om_delete*( ) directly on public subobjects (for example, 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 (
    const OM_private_object             original,
    const 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 (
    const OM_private_object           original,
    const OM_exclusions               exclusions,
    const OM_type_list                included_types,
    const OM_boolean                  local_strings,
    const OM_value_position           initial_value,
    const OM_value_position           limiting_value,
    OM_public_object                 *copy,
    OM_value_position                *total_number
);
```

DESCRIPTION

This operation creates a new public object that is a 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. The copy is an independent copy unless the use of **exclude-subobjects** results in the sharing of subobjects.

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: 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.

- **exclude-all-but-these-types**
  The copy includes descriptors encompassing only attributes of specified types. This exclusion provides a means for determining the values of specified attributes, as well as the syntaxes of those values.

- **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.

  This exclusion provides a means for discerning the presence of multi-valued attributes without simultaneously getting their values.

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

  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.

- **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.

  This exclusion provides a means for determining an objects composition, that is, the type and syntax of each of its attribute values.

- **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.

  This exclusion provides a means for examining an object one *level* at a time.

- **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.

  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: 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)
Relevant if, and only if, the **exclude-all-but-these-types** exclusion is requested, in which case it 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)
Relevant if, and only if, the **exclude-all-but-these-values** exclusion is requested, in which case it identifies 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)
Relevant if, and only if, the **exclude-all-but-these-values** exclusion is requested, in which case it identifies 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: 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, 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 (
    const OM_object                  subject,
    const 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: 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,
    const OM_modification          modification,
    const OM_object                source,
    const OM_type_list             included_types,
    const OM_value_position        initial_value,
    const 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. Only those attributes that pertain to the destination object are copied, attributes that are not relevant are ignored. 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:

- **insert-at-beginning**
  The source values are inserted before any existing destination values. (The latter are retained.)
- **insert-at-certain-point**
  The source values are inserted before the value at a specified position in the destination attribute. (The latter are retained.)
- **insert-at-end**
  The source values are inserted after any existing destination values. (The latter are retained.)
- **replace-all**
  The source values are placed in the destination attribute. The existing destination values, if any, are discarded. (The latter are discarded.)
- **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)
Identifies the types of the attributes to be included in the destination (provided that they appear in the source); if the list is empty, or if (in the C interface) the argument is a NULL pointer, then all attributes are to be included.

Initial Value (Value Position)
Relevant if, and only if, the Modification argument is **insert-at-certain-point** or **replace-certain-values**, in which case it identifies the position within each destination attribute at which source values are to be inserted, or of the first value to be replaced, respectively.

If the Modification argument is **insert-at-certain-point** and if the Initial Value argument is **all-values** or exceeds the number of values present in a destination attribute, the argument is taken to be equal to the number of values present.

Limiting Value (Value Position)
Relevant if, and only if, the Modification argument is **replace-certain-values**, in which case it identifies 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 (
    const OM_private_object       subject,
    const OM_type                 type,
    const OM_value_position       value_position,
    const OM_boolean              local_string,
    const 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: 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 (
    const OM_private_object      subject,
    const OM_type                type,
    const OM_value_position      initial_value,
    const 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 (
    const OM_private_object      subject,
    const OM_type                type,
    const OM_value_position      value_position,
    OM_syntax                    syntax,
    const 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: 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 not 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.

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 *Staring 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 the following table 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 on page 58.

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 in Section 4.3 on page 34.

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	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
wrong-value-type	-	x	-	x	-	-	x	-	x	-	-	-

Table 4-3 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 (for example, a keystroke, designated for this purpose, at a user interface).

memory-insufficient
The service cannot allocate the main memory it needs to complete the function.

network-error
The service could not successfully employ the network upon which its implementation depends.

no-such-class
A purported class identifier is undefined.

no-such-exclusion
A purported exclusion identifier is undefined.

no-such-modification
A purported modification identifier is undefined.

no-such-object
A purported object is nonexistent or the purported handle is invalid.

no-such-rules
A purported rules identifier is undefined.

no-such-syntax
A purported syntax identifier is undefined.

no-such-type
A purported type identifier is undefined.

no-such-workspace
A purported workspace is nonexistent.

not-an-encoding
An object is not an instance of the Encoding class.

not-concrete
A class is abstract, not concrete.

not-present
An attribute value is absent, not present.

not-private
An object is public, not private.

not-the-services
An object is client-generated, rather than service-generated or private.

permanent-error
The service encountered a permanent difficulty other than those denoted by other return codes.

pointer-invalid
In the C interface, an invalid pointer was supplied as a function argument or as the receptacle for a function result.

system-error
The service could not successfully employ the operating system upon which its implementation depends.

temporary-error
The service encountered a temporary difficulty other than those denoted by other return codes.

too-many-values
An implementation limit prevents the addition to an object of another attribute value. This limit is undefined.

values-not-adjacent
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 on page 21, and

The function macros that appear (in Section 5.3 on page 67) in the specification of the workspace interface here replace the function prototypes that appear (in Section 4.3 on page 34) 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). 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;
typedef system-defined, e.g., double                OM_double;

/* 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;

/* Real */
typedef OM_double OM_real;

/* 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_real            real;
} OM_value;

/* macro to extract an object from a value */

#define OM_OBJ_VALUE(v)        ((OM_object)((v).string.elements))

/* Macro to extract syntax from a descriptor */

#define OM_SYNTAX(d)           ((d).syntax & OM_S_SYNTAX)
```

```
/* Macro to assign a syntax to a descriptor */

#define OM_SYNTAX_ASSIGN(d,s)   ((d).syntax=(s)|((d).syntax & ˜OM_S_SYNTAX))

/* Macro to extract additional information contained in the syntax */

#define OM_HAS_VALUE(d)             (!((d).syntax & OM_S_NO_VALUE))
#define OM_IS_PRIVATE(d)            ((d).syntax & OM_S_PRIVATE)
#define OM_IS_SERVICE_GENERATED(d)  ((d).syntax & OM_S_SERVICE_GENERATED)
#define OM_IS_LOCAL_STRING(d)       ((d).syntax & OM_S_LOCAL_STRING)
#define OM_IS_LONG_STRING(d)        ((d).syntax & OM_S_LONG_STRING)

/* Macro to set additional information contained in the syntax */

#define OM_SET_LOCAL_STRING(d)      ((d).syntax |= OM_S_LOCAL_STRING)

/* 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_DESCRIPTORS            ( (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             "\x2A\x86\x48\xCE\x21\x00"

/* Constant for the Encoding class                                  */

#define OMP_O_OM_C_ENCODING     "\x2A\x86\x48\xCE\x21\x00\x01"

/* Constant for the External class                                  */

#define OMP_O_OM_C_EXTERNAL     "\x2A\x86\x48\xCE\x21\x00\x02"

/* Constant for the Object class                                    */

#define OMP_O_OM_C_OBJECT       "\x2A\x86\x48\xCE\x21\x00\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  "\x2A\x86\x48\xCE\x21\x00\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_REAL                        ( (OM_syntax) 9 )
#define OM_S_TELETEX_STRING              ( (OM_syntax) 20 )
#define OM_S_UNIVERSAL_STRING            ( (OM_syntax) 28 )
#define OM_S_UNRESTRICTED_STRING         ( (OM_syntax) 29 )
#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 */
```

Chapter 5

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
- 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. These are referred to as the *-1st* and *0th* elements.

The external representation (that is, the pointer returned to the client by *om_create*( ), etc.) points to the second, *0th*, descriptor. The client is not aware of the existence of the first, *-1st*, descriptor.

Note: 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.

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 (that is, **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.

Notes:

1. 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.
2. 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.

Notes:

1. 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.
2. 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.

Each macro must check if the object is valid (different from NULL) and is a private object. If not, the macro must return the corresponding error code.

```
#include <stddef.h>

#define om_copy(ORIGINAL,WORKSPACE,COPY)                                    \
    ( (WORKSPACE) == NULL   ?                                               \
      OM_NO_SUCH_WORKSPACE   :                                              \
       (((OMP_workspace)(WORKSPACE) )->functions->omp_copy(                 \
             (ORIGINAL),(WORKSPACE),(COPY) ))                               \
    )

#define om_create(CLASS,INITIALISE,WORKSPACE,OBJECT)                        \
    ( (WORKSPACE) == NULL   ?                                               \
      OM_NO_SUCH_WORKSPACE   :                                              \
       (((OMP_workspace)(WORKSPACE) )->functions->omp_create(               \
               (CLASS),(INITIALISE),(WORKSPACE),(OBJECT) ))                 \
    )

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

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

#define om_copy_value(SOURCE,  SOURCE_TYPE,  SOURCE_POSITION,               \
                   DEST,    DEST_TYPE,    DEST_POSITION)                    \
    ( (DEST) == NULL        ?                                               \
      OM_NO_SUCH_OBJECT     :                                               \
      (((DEST)->syntax  & OM_S_PRIVATE)      ?                              \
        (OMP_FUNCTIONS(DEST)->omp_copy_value(                               \
                   (SOURCE),  (SOURCE_TYPE),  (SOURCE_POSITION),            \
                   (DEST),    (DEST_TYPE),    (DEST_POSITION))):            \
        OM_NOT_PRIVATE                                                      \
      )                                                                     \
    )
```

```
#define om_decode(ENCODING,ORIGINAL)                                      \
    ( (ENCODING) == NULL    ?                                             \
      OM_NO_SUCH_OBJECT     :                                             \
      (((ENCODING)->syntax  & OM_S_PRIVATE)  ?                            \
        (OMP_FUNCTIONS(ENCODING)->omp_decode( (ENCODING),(ORIGINAL) )):  \
        OM_NOT_PRIVATE                                                    \
      )                                                                   \
    )

#define om_encode(ORIGINAL,RULES,ENCODING)                                \
    ( (ORIGINAL) == NULL    ?                                             \
      OM_NO_SUCH_OBJECT     :                                             \
      (((ORIGINAL)->syntax  & OM_S_PRIVATE)  ?                            \
         (OMP_FUNCTIONS(ORIGINAL)->omp_encode(                            \
                           (ORIGINAL),(RULES),(ENCODING) )) :             \
        OM_NOT_PRIVATE                                                    \
      )                                                                   \
    )

#define om_get(ORIGINAL,EXCLUSIONS,TYPES,LOCAL_STRINGS,                   \
                    INITIAL,LIMIT,COPY,TOTAL_NUMBER)                      \
    ( (ORIGINAL) == NULL    ?                                             \
      OM_NO_SUCH_OBJECT     :                                             \
      (((ORIGINAL)->syntax  & OM_S_PRIVATE)  ?                            \
         (OMP_FUNCTIONS(ORIGINAL)->omp_get(                               \
            (ORIGINAL),(EXCLUSIONS),(TYPES),(LOCAL_STRINGS),              \
                    (INITIAL),(LIMIT),(COPY),(TOTAL_NUMBER) )):           \
        OM_NOT_PRIVATE                                                    \
      )                                                                   \
    )

#define om_put(DESTINATION,MODIFICATION,SOURCE,TYPES,INITIAL,LIMIT)       \
    ( (DESTINATION) == NULL ?                                             \
      OM_NO_SUCH_OBJECT     :                                             \
      (((DESTINATION)->syntax  & OM_S_PRIVATE)       ?                    \
        (OMP_FUNCTIONS(DESTINATION)->omp_put(                             \
            (DESTINATION),(MODIFICATION),(SOURCE),(TYPES),                \
               (INITIAL),(LIMIT) ))                          :            \
        OM_NOT_PRIVATE                                                    \
      )                                                                   \
    )

#define om_read(SUBJECT,TYPE,VALUE_POS,LOCAL_STRING,                      \
                    STRING_OFFSET,ELEMENTS)                               \
    ( (SUBJECT) == NULL     ?                                             \
      OM_NO_SUCH_OBJECT     :                                             \
      (((SUBJECT)->syntax  & OM_S_PRIVATE)   ?                            \
         (OMP_FUNCTIONS(SUBJECT)->omp_read(                               \
          (SUBJECT),(TYPE),(VALUE_POS),(LOCAL_STRING),                    \
            (STRING_OFFSET),(ELEMENTS) ))                    :            \
        OM_NOT_PRIVATE                                                    \
      )                                                                   \
    )
```

```
#define om_remove(SUBJECT,TYPE,INITIAL,LIMIT)                              \
    ( (SUBJECT) == NULL     ?                                              \
      OM_NO_SUCH_OBJECT     :                                              \
      (((SUBJECT)->syntax & OM_S_PRIVATE)   ?                              \
         (OMP_FUNCTIONS(SUBJECT)->omp_remove(                              \
              (SUBJECT),(TYPE),(INITIAL),(LIMIT) ))          :             \
        OM_NOT_PRIVATE                                                     \
      )                                                                    \
    )

#define om_write(SUBJECT,TYPE,VALUE_POS,SYNTAX,STRING_OFFSET,ELEMENTS)   \
    ( (SUBJECT) == NULL     ?                                              \
      OM_NO_SUCH_OBJECT     :                                              \
      (((SUBJECT)->syntax & OM_S_PRIVATE)   ?                              \
         (OMP_FUNCTIONS(SUBJECT)->omp_write(                               \
            (SUBJECT),(TYPE),(VALUE_POS),                                  \
            (SYNTAX),(STRING_OFFSET),(ELEMENTS) ))           :             \
        OM_NOT_PRIVATE                                                     \
      )                                                                    \
    )
```

Chapter 6

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

```
[iso(1) member-body(2) us(840) IEEE-1224(10017) om(0)]
```

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 (for example, 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 the following table.

	Value	**Value**	**Value**	**Value**
Attribute	**Syntax**	**Length**	**Number**	**Initially**
Object Class	String(Object Identifier)	-	0-1	-
Object Encoding	String[1]	-	1	-
Rules	String (Object Identifier)	-	1	ber

[1] The syntax of this attribute is determined by the requirements of the encoding rules used to perform the encoding. If the Rules attribute is ber or canonical-ber, the syntax is `String(Encoding)`.

Table 6-1 Attributes Specific to Encoding

Object Class
Identifies the class of the object that the Object Encoding attribute encodes. The service shall provide a value for this attribute if and only if the class is represented in one of the packages that the client has negotiated, in the workspace of the Encoding object. The client may but need not, provide a value for this attribute. The class, if present, 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**.
  This value denotes the BER (Clause 25.2 of Recommendation X.208 - see references **ASN.1** and **BER**). It is specified in ASN.1 as

```
{joint-iso-ccitt asn1(1) basic-encoding(1)}
```

- **canonical-ber**.
  This value denotes the canonical BER (Clause 8.7 of Recommendation X.509 - see reference **X.509**). It is specified in ASN.1 as

```
{joint-iso-ccitt mhs-motis(6) group(6) white(1) api(2) om(4) \
                                               canonical-ber(4)}
```

Note: 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 reference **ASN.1**).

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

Attribute	Value Syntax	Value Length	Value Number	Value Initially
Arbitrary Encoding	String (Bit)	-	0-1[1]	-
ASN1 Encoding	String (Encoding)	-	0-1[1]	-
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[1]	-

[1]Exactly one of these three attributes shall be present.

Table 6-2 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 reference **ASN.1**).

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 reference **ASN.1**).

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 reference **ASN.1**).

Direct Reference
A direct reference to the data type. This attribute is described more fully in Clause 34 of Recommendation X.208 (see reference **ASN.1**).

Indirect Reference
An indirect reference to the data type. This attribute is described more fully in Clause 34 of Recommendation X.208 (see reference **ASN.1**).

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 reference **ASN.1**).

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 the following table.

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

Table 6-3 Attributes Specific to Object

Class
Identifies the object's class.

Appendix A

Differences from IEEE OM Standard

The IEEE has published its **Object Management (OM)** Standard. It is published as IEEE 1224-1993 (language independent standard) and IEEE 1327-1993 (C binding). Development of this IEEE Standard was based on the X/Open **OSI-Abstract-Data Manipulation (XOM)** API CAE Specification (November 1991).

The intent of this **Issue 2** of X/Open's **OSI-Abstract-Data Manipulation (XOM) API** CAE Specification is to align with the corresponding IEEE OM Standard wherever possible. However, there are a few instances where full alignment between the IEEE 1224/1327 Standard and the X/Open XOM API CAE Specification has not been achieved.

This Appendix identifies the known substantive differences between this X/Open **XOM API CAE Specification Issue 2**, and the corresponding **IEEE Object Management Standard**.

A.1 Copy-Value( ) Clarification

In the Description section of the **Copy-Value** function, the **XOM** specification states:

> If the value of the destination attribute exists, the resulting value shall have the syntax of the replaced value. If the source valur does not have the same syntax as the replaced value, the error code wrong-value-syntax shall be returned.
>
> If the value of the destination attribute does not exist, the new value shall have the syntax of the source value. If the syntax of the source value is not compatible with the description of the destination attribute, the error code wrong-value-syntax shall be returned.

The corresponding description in IEEE 1224-1993 does not state that the error code wrong-value-syntax shall be returned when the syntax of the source does not satisfy the description of the destination object. It is arguable, however, that the description of the wrong-value-syntax error code in IEEE 1224-1993 implies that this must be the case. Subject to a favourable interpretation of this point by the IEEE, this is an instance of a clearer statement in the XOM specification, rather than a substantive difference between the specifications.

A.2 Meaning of Value Length Column

IEEE 1224-1993 states that the **Value Length** column of each class definition table indicates any constraints on the number of octets in a string attribute, whereas XOM states that that the indicated constraints are on the numbers of bits, octets or characters.

X/Open believes that the definition in IEEE 1224-1993 is inconsistent with other definitions in IEEE 1224-1993, and is wrong.

A.3 Meaning of Private Bit

IEEE 1327 states that the OM_S_PRIVATE bit is 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. XOM states that it is 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.

The difference between these definitions is substantive. X/Open believes that the IEEE definition is unworkable.

A.4 Constant Values Optional

The numerical values of constants in the **#define** values in Section 4.5 on page 58 (Declaration Summary) and the **#define** values, structures and macros defined in Chapter 5 (Workspace Interface) are mandatory in XOM but are optional in IEEE 1327-1993.

This means that XOM is more restrictive than IEEE 1224-1993. An implementation that conforms to XOM in this respect will therefore conform to IEEE 1224-1993 also. However, an implementation that conforms to IEEE 1224-1993 need not conform to XOM. A well-behaved application (that is, one which uses the **#define** symbols rather than their numerical equivalents) will not be affected by this difference.

A.5 Deletion of SPUBs on Workspace Closure

IEEE 1224-1993 states that a service-generated public object associated with a workspace shall be destroyed when the workspace containing it is destroyed, but XOM states that a service generated public object is unaffected by the destruction of the workspace that generated it.

This is a substantive difference between the two specifications. X/Open believes that some applications of the X.400 API (see reference **X.400**) require SPUBS to be preseved on workspace destruction, and therefore decided not to align XOM with IEEE 1224-1993.

A.6 Meaning of ‘‘present’’

In IEEE 1224-1993, the meaning of the word *present* in the descriptions of the *Included Types*, *Initial Value* and *Limiting Value* arguments of *om_get*( ) and *om_put*( ) is unclear.

In XOM, the word *present* is replaced by *relevant* in the description of the *Initial Value* and *Limiting Value* arguments of *om_get*( ) and *om_put*( ), and in the *Included Types* argument of *om_get*( ), and further minor wording changes consistent with this have been made. The description of the *Included Types* argument of *om_put*( ) in XOM has been changed to:

> Identifies the types of the attributes to be included in the destination (provided that they appear in the source); if the list is empty, or if (in the C interface) the argument is a NULL pointer, then all attributes are to be included.

Subject to a favourable interpretation of this point by the IEEE, this is an instance of a clearer statement in the XOM specification, rather than a substantive difference between the specifications.

A.7 Addition of a REAL data syntax

A new REAL data syntax has been added to this issue of the XOM specification. This represents a difference between this XOM API and the API defined by the IEEE 1224-1993 standard.

A.8 Creation of Restricted or Abstract Classes

In a previous issue of the XOM specification, the client of an API was not allowed to create an abstract class or one for which the definition states that a client may not create instances. In the present issue, this restriction has been removed. This represents a difference between this XOM API and the API defined by the IEEE 1224-1993 standard.

A.9 Definition of Dispatcher Macros

In this issue of the XOM specification, the definition of the dispatcher macros has been modified to ensure that the object or workspace handle is not NULL and that an object is service-generated. This change is intended to trap a potential core dump if an invalid handle is passed to these macros. This additional error checking represents a difference between this XOM API and the API defined by the IEEE 1224-1993 standard.

A.10 Representation of OPTIONAL ASN.1 Constructs

In aligning this issue of the XOM specification with the IEEE 1224-1993 standard, the convention for the representation of OPTIONAL ASN.1 constructs has been made advisory (should) rather than mandatory (shall). This allows packages that were defined using the convention defined by a previous version of the XOM specification to continue to comply with the requirements of this specification and maintains the advisory spirit of the text. This represents a difference between this XOM API and the API defined by the IEEE 1224-1993 standard.

A.11 Support of Internationalised Character Strings

Two new character string syntaxes (`Universal String` and `Unrestricted String`) have been added to this issue of the XOM specification, to support internationalised client applications. This represents a difference between this XOM API and the API defined by the IEEE 1224-1993 standard.

Glossary

Abstract
Said of a class, instances of which are forbidden.

Accessible
Said of an object for which the client possesses a valid designator or handle.

Attribute
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.

Bit String
A string comprising bits.

C Interface
The interface, defined at a level that depends upon the variant of C standardised by ANSI.

Character String
A string comprising characters.

Class
A category into which objects are placed on the basis of both their purpose and their internal structure.

Client
Software that uses the interface.

Concrete
Said of a class, instances of which are permitted.

Descriptor
The means by which the client and service exchange an attribute value and the integers that denote its representation, type and syntax.

Dispatcher
The software that implements the service interface functions using workspace interface functions.

Element
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.

Generic Interface
The interface, defined at a level that is independent of any particular programming language

Immediate Subclass
A subclass, of a class C, having no superclasses that are themselves subclasses of C.

Immediate Subobject
One object that is a value of an attribute of another.

Immediate Superclass
The superclass, of a class C, having no subclasses that are themselves superclasses of C.

Immediate Superobject
One object that contains another among its attribute values.

Inaccessible
Said of an object for which the client does not possess a valid designator or handle.

Instance
An object in the category represented by a class.

Interface
The interface this document defines.

Intermediate Data Type
Any of the basic data types in terms of which the other, substantive data types of the interface are defined.

Intermediate Macro
In the C interface, any of the basic macros in terms of which the other, substantive macros used to realise the dispatcher are defined.

Length
The number of elements in a string.

Minimally Consistent
Said of an object that satisfies various conditions set forth in the definition of its class.

Object Management
The creation, examination, modification and deletion of potentially complex information objects.

Object
Any of the complex information objects created, examined, modified or destroyed by means of the interface.

Octet String
A string comprising octets.

OSI Object Management API
The interface this document defines.

Package
A group of related classes.

Package Closure
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.

Position (within a string)
The ordinal position of one element of a string relative to another.

Position (within an attribute)
The ordinal position of one value relative to another.

Primary Representation
The form in which the service supplies an attribute value to the client.

Private Object
An object that is represented in an unspecified fashion.

Public Object
An object that is represented by a data structure whose format is part of the service's specification.

Secondary Representation
A second form, an alternative to the primary representation, in which the client may supply an attribute value to the service.

Segment
Zero or more contiguous elements of a string.

Service Interface
The interface as realised, for the client's benefit, by the service as a whole.

Service
Software that implements the interface.

Specific
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.

String
An ordered sequence of zero or more bits, octets or characters, accompanied by the string's length.

Subclass
One of the classes, designated as such, whose attribute types are a superset of those of another class.

Subobject
An immediate subobject of an object or of one of its subobjects.

Superclass
One of the classes, designated as such, whose attribute types are a subset of those of another class.

Superobject
An object's immediate superobject, or one of its superobjects.

Syntax template
A lexical construct containing an asterisk from which several attribute syntaxes can be derived by substituting text for the asterisk.

Syntax
A category into which an attribute value is placed on the basis of its form.

Type
A category into which attribute values are placed on the basis of their purpose.

Value
An arbitrarily complex information item that can be viewed as a characteristic or property of an object.

Workspace Interface
The interface as realised, for the dispatcher's benefit, by each workspace individually.

Workspace
A repository for instances of classes in the closures of one or more packages associated with the workspace.

Index