Solstice™ XOM Programming Reference

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A

Part No.: 802-1311-12
Revision A, February 1996

Please
Recycle

Adobe PostScript

Contents

Figures

Tables

Preface

This document includes text derived from the original "*OSI-Abstract-Data Manipulation (XOM) API*" document with the permission of X/Open Company Limited. The text in this document is not the definitive version of the specification. The definitive version may be purchased from X/Open Company Limited, which takes no responsibility for the accuracy of this text.

Nothing in this document shall imply that the text of that carries the authority of X/Open or that it may be used as the basis for establishing the conformance of any product.

Who Should Use This Book

This document is written for the person with a working knowledge of XOM principles and concepts. Code segments are provided to facilitate implementation of X/Open Object Management (XOM).

How This Book Is Organized

The *Solstice XOM Programming Reference* is organized as follows:

Chapter 1, "Introduction," introduces the interface specification and conformances issues in the XOM development environment.

Chapter 2, "Information Architecture," covers the client and service exchange as specified in objects communication architecture.

Chapter 3, "Information Syntax," defines the acceptable syntaxes required for the attribute values using ASN.1 specifications.

Chapter 4, "Service Interface," describes the function interface services that are used by the client, data types, arguments, and return values.

Chapter 5, "Workspace Interface," explains the workspace interface that specifies the representation of objects and associated data structures.

Chapter 6, "Object Management Package," covers the symbolic object identifiers that are assigned to the various packages.

Appendix A, "Referenced Documents," list the associated documents for XOM development environment.

Glossary lists specific terminology used in XOM development.

Related Books

Refer to Appendix A, "Referenced Documents," for more specific information on other related documents.

Examples

The use of the *OSI-Abstract-Data Manipulation API* is illustrated in the examples provided on the product CD-ROM.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Abbreviations

The following abbreviations are used in this document.

Table P-3 Abbreviations

Abbreviation	**Meaning**
API	Application Program Interface
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CCITT	International Telegraph and Telephone Consultative Committee
IA5	International Alphabet No. 5
ISO	International Organization for Standardization
OM	Object Management
OSI	Open Systems Interconnection
UCT	Universal Coordinated Time

Introduction 1

This chapter introduces the object management interface and its specification. It indicates the purpose and motivation of the interface. The different levels of abstraction within the interface are defined with explanations of how one level differs from another. A summary of the service implementation options, a listing of abbreviations, and conformance requirements is also included.

1.1 Purpose

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

Object Management (OM) is the creation, examination, modification and deletion of potentially complex information objects. The architecture model implements a concept of groups or classes of similar objects.

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 consist of tools for manipulating ASN.1 objects. It shields the programmer from much of ASN.1's complexity.

The OM API is an open architecture; thereby, allowing multiple vendors to implement their specific applications independently. Figure 1-1 illustrates how each client provides the developer with the ability to manipulate information objects of particular type.

This division of implementation is achieved through the use of workspaces. A complete explanation can be found in “Workspace” on page 2-9.

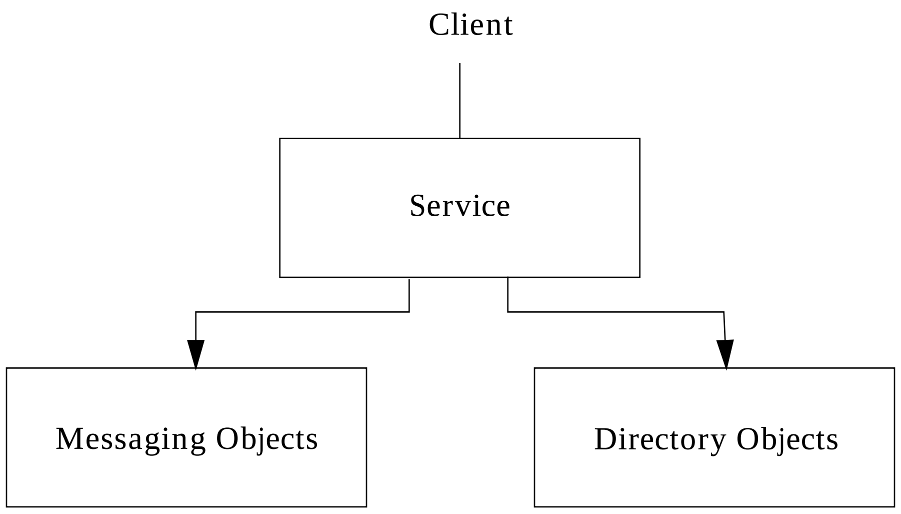

Figure 1-1 Conceptual Model of Object Management

Throughout this document, the term interface denotes the OM API. The term *service* denotes software that implements the interface, while *client* denotes software that uses the interface. The *service interface* denotes the interface used by the service, also it is used as a synonym for *interface*.

1.2 Reasons for the OM Interface

The OM interface is designed to be used with other application-specific APIs, such as message transfer or directory access. It defines functions and structured information objects that serve as arguments to functions. 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 employs modularity and extensibility. A collection of data are referred to as objects, similar objects in the structure are grouped together into classes.

Classes are related to one another by subclassing. The information architecture is described in detail in Chapter 2, “Information Architecture”. An instance of a class and instances of that class’ subclasses are important features for an application-specific API. It permits the refinement, extension of objects and the use of extended objects in application-specific APIs.

Note, while the 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. However, the interface does hide information representation.

The OM interface is specifically designed for use with application-specific APIs that provide OSI services. The objects addressed by the information architecture are those arising from ASN.1 used 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 the OM interface presents to the client a single model of information, it 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 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 a sequence 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 “Workspace” on page 2-9).

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 recognizes 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 of Abstraction

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 standardized 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, the specifications of the generic and C interfaces are combined.

1.4 C Naming Conventions

How the identifier for an element of the C interface is derived from the name of the corresponding element of the generic interface depends on the element's type, as specified in Table 1-1. 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 ( ).

Table 1-1 Derivation of C Identifiers

Element Type	**Prefix**	**Case**
Data type	OM_	Lower
Data value	OM_	Upper
Data value (Class)	OM_C_	Upper
Data value (Syntax)	OM_S_	Upper

Table 1-1 Derivation of C Identifiers

Element Type	**Prefix**	**Case**
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

The prefixes “omP” and “OMP” are reserved for implementors. 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 behavior are defined by implementation:

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

Information Architecture 2

This chapter covers the information architecture available through the service interface.

2.1 Introduction

The architecture specifies how the client communicates with the service, client response to service communication, and service communication components. The architecture does not dictate the physical structure of information because the service maintains it internally which is implementation-specific. Refer to Chapter 4, “Service Interface” and Chapter 5, “Workspace Interface”, for additional information.

2.2 Objects

The purpose of the service is to create, examine, modify and delete complex information objects under the client’s direction. The purpose of the interface is to enable the client and service to exchange objects. This requirement provides the rationale for the information architecture.

There are two kinds of objects: *public* and *private*. A *public object* is represented by a data structure whose format is part of the service specification (refer to “Data Types” on page 4-1). A *private object* is represented in a fashion that is implementation-specific and is unspecified. Therefore, the client can access private objects indirectly through the interface functions.

The interface consists of functions that examine and modify private objects. For application-specific reasons the service may deny a client request to modify a particular object at a particular time. The specification of each application-specific API identifies any circumstances under which this may occur.

There are two kinds of public objects: *client-generated* and *service-generated*. A client-generated public object is constructed by the client in storage it provides. A service-generated public object is constructed by the service in storage it provides. The client can create, examine, modify, and destroy client-generated public object using programming language constructs.

Note, client-generated public objects simplify application programs that allow them to define objects statically when appropriate. This is the reverse of dynamic construct where the objects use a sequence of interface function calls.

2.2.1 Object Attributes

Objects have internal structure, as illustrated in Figure 2-1. An object consist of zero or more information items called attributes. An *attribute*, is an integer denoting the attribute's type and one or more information items called values. Each attribute is accompanied by an integer denoting that value's syntax. A *value* is an information item, possibly complex, which can be viewed as a character or property of the object of which it is a part. A *syntax* is a category where a value is placed on the basis of its form. A *type* is a category where 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.

Note that Figure 2-1 shows a simple structure with only one attribute. Objects can have more than one attribute.

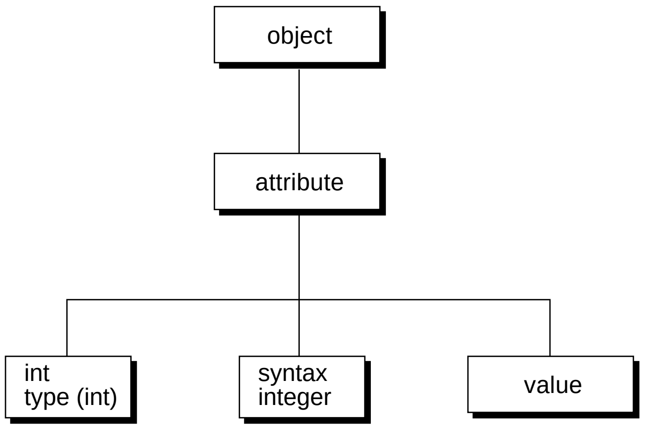

Figure 2-1 Structure of an Object

The client and service exchange values through 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. Refer to "Packages" on page 2-8 for additional information.

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, O2, may be a value of an attribute of another object, O1. O2 is called an *immediate subobject* of O1, and O1 the *immediate superobject* of O2. The immediate subobjects of O1, and all of their subobjects, are the *subobjects* of O1. The immediate superobject of O2, and all of its superobjects, are the superobjects of O2. The package that contains an object's class may differ from those containing the classes of its immediate subobjects, which may differ from one another.

2.3 Classes

Objects are categorized 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. A class is characterized 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 can be assigned to a class in two steps. First, a distinct integer is assigned to each class in a package. Second, the integer is appended to the object identifier assigned to the package, becoming its final sub-identifier.

The types that may appear in an instance of one class, C2, are often a superset of those that may appear in an instance of another class, C1. When this is so, C2 may (but need not) be designated a *subclass* of C1, making C1 a *superclass* of C2. If C1 is a superclass of no other superclass of C2, C1 is called the *immediate superclass* of C2, and C2 an *immediate subclass* of C1. Every class (except object) is the immediate subclass of exactly one other class; thus the class hierarchy is a tree.

Figure 2-2 shows that every inheritance hierarchy begins with a root class that has no superclass. At the root class, the hierarchy branches downward. Each class inherits from its superclass and through its superclass, from all the classes above it in the hierarchy. Every class inherits from the root class.

Each new class is the accumulation of all the class definition in its inheritance chain. For example, class D inherits both from C, its superclass and the root class. Members of D class will have methods and instance variables defined in all three classes—D, C, and root.

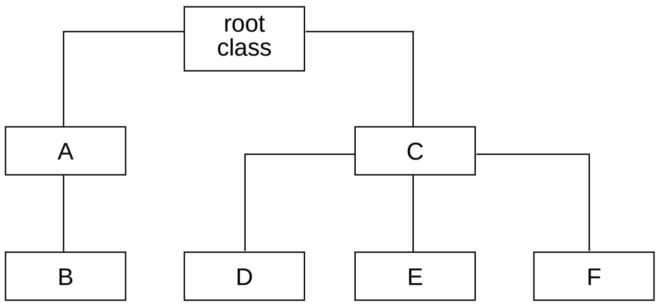

Figure 2-2 Class Inheritance

The package 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.

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 "Object" on page 6-5. The class hierarchy is fixed by the class definitions and cannot be altered programmatically. Refer to Figure 2-2.

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 "Class Definitions" on page 2-6); it cannot be altered programmatically. The fact that an attribute may appear in instances of a class does not imply that it must appear that is, some attributes are optional.

An instance of a class is also considered an instance of each of its superclasses, and may appear wherever the interface requires an instance of any of those classes. This is one of the most useful consequences of the subclass mechanism.

There are two types of classes: *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, an instance can only be created as a result of an application-specific function. It is an error for a client to create an object of such a class (*function-declined*), or an abstract class (*not-concrete*).

The OM information architecture has some, but not all, of the important characteristics of object-oriented programming systems. The functions by means of objects are manipulated, for example, may vary from workspace to workspace, but not from class to class. Refer to “Workspace” on page 2-9.

2.3.1 Class Definitions

For purposes of the generic interface, and within the context provided by a package 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 attributes specific to a class are defined in a table. For each attribute, the following are defined:

- **Attribute** is the name of each attribute
- **Value Syntax** is the syntax or syntaxes of each value
- **Value Length** indicates any constraint upon the number of bits, octets or characters in each value that is a string
- **Value Number** indicates any constraints upon the number of values
- **Value Initially** indicates any value the `Create` function supplies upon request

Table 2-1 shows the definition of the attributes specific to the Encoding class.

Table 2-1 Attributes Specific to Encoding

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

[1] If the Rules attribute is **ber** or **canonical-ber**, the syntax of the present attribute shall be String (Encoding).

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

A class table imposes certain constraints upon instances of the class. The definition of a class may impose additional constraints which can be arbitrarily complex, such constraints are specified.

Classes are often constrained in additional ways. Each instance of the class may be constrained to contain exactly one member of a set of attributes. An attribute can 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. The implementation determines if the service reports an exception or not.

2.4 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 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 that package.

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 is a set that consists of:

- The class itself
- The closures of any subclasses defined in the same package
- The closures of the classes of all permitted subobjects of instances that class

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

2.4.1 Package Definitions

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

- The package name denotes its object identifier
- The definitions of the one or more classes which make up the package
- The identification of zero or more concrete classes in the package to which the `Create` function applies in every implementation of the service.
- The identification of zero or more concrete classes in the package to which the `Encode` function applies in every implementation of the service.
- 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.

2.5 Workspace

Two application-specific APIs may involve the same class and may employ different implementations of the service. For example, different vendors may represent private objects differently. If they both use the same API, 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 object's classes are associated with both. Note, it is outside the scope of this document how workspaces are created, made known to the client, and destroyed. In all cases, destroying a workspace effectively applies the `Delete` function to each private object it contains.

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. 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.

2.6 Storage Management

An object occupies storage. Both public and private objects occupies main and secondary storage or a combination of the two. This is an option of the workspace where 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 through an object handle. An object handle is way the client supplies an object to the service as an argument of an interface function. 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, that is, 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 behavior is undefined.

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

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

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

Information Syntax 3

This chapter defines the permitted syntax of attribute values. The syntaxes are highly aligned with the type constructors of ASN.1. Information on how a value of each syntax is represented in the C interface is specified by the value data type (refer to the “Data Types” on page 4-1).

3.1 Syntax Templates

The names of certain syntaxes are constructed from syntax templates. A syntax template comprises a primary identifier followed by an asterisk enclosed in parentheses: *identifier(*).*

A syntax template is a group of related syntaxes. Any member of the group, without distinction, is denoted by the primary *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: *identifier1 (identifier2).*

3.1.1 Syntaxes

A variety of syntaxes are defined. Most are functionally equivalent to ASN.1 types, as explained in “Relationship to ASN.1 Simple Types” on page 3-4.

The following syntaxes are defined:

Boolean
A value of this syntax is true or false.

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 an integer.

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

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

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

String (*)
A value of any syntax encompassed by this syntax template is a string (as defined in “Strings” on page 3-2) 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.

3.2 Strings

A *string* is an ordered sequence of zero or more bits, octets or characters. A string is categorized 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, 232). Any constraints on the value length of a string are specified in the appropriate class definitions.

The syntaxes that form the String group are identified in Table 3-1, 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.

Table 3-1 String Syntax Identifiers

Bit String Identifier	Octet String Identifier	Character String Identifier
Bit	Encoding[1]	General[2]
	Object Identifier[3]	Generalized Time[2]
	Octet	Graphic[2]
		IA5[2]
		Numeric[2]
		Object Descriptor[2]
		Printable[2]
		Teletex[2]
		UTC Time[2]
		Videotex[2]
		Visible[2]

[1] The octets are those the BER permit for the contents octets of the encoding of a value of any ASN.1 type.

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

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

3.2.1 Representation of String Values

In the service interface, a string value is represented by the string data type. This is defined in "Strings" on page 3-2. The length of a string is the number of octets by which it is represented at the interface. It is confined to the interval $[0, 2^{32})$.

Note – The length of a character string may not equal the number of characters it comprises, 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.

3.3 Relationship to ASN.1 Simple Types

As shown in Table 3-2, 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.

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

Type	**Syntaxes**
Bit String	String (Bit)
Boolean	Boolean
Integer	Integer
Null	Null
Object Identifier	String (Object Identifier)
Octet String	String (Octet)
Real	Real[1]

[1] Implemented only for the Solstice CMIP product.

3.4 Relationship to ASN.1 Useful Types

As shown in Table 3-3, 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.

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

Type	**Syntax**
External	Object (External)
Generalized Time	String (Generalized Time)
Object Descriptor	String (Object Descriptor)
Universal Time	String (UTC Time)

3.5 Relationship to ASN.1 Character String Types

As shown in Table 3-4, 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.

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

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

3.6 Relationship to ASN.1 Type Constructors

As shown in Table 3-5, 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.

Table 3-5 Syntaxes for ASN.1's Type Constructors

Type	**Syntax**
Any	String (Encoding)
Choice	Object[1]
Enumerated	Enumeration
Selection	*none*[2]
Sequence	Object[1]
Sequence Of	Object[1]

Table 3-5 Syntaxes for ASN.1's Type Constructors

Type	**Syntax**
Set	Object[1]
Set Of	Object[1]
Tagged	*none*[3]

[1] See the text.

[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.

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. If an element is OPTIONAL then the attribute should have zero or one values.

Sequence Of or Set Of
A single, multi-valued attribute may be defined.

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

Service Interface 4

This chapter defines the service interface. Specific information about functions that the service makes available to the client, arguments to data types, data values, return codes, and C service interface is also provided.

4.1 Data Types

This section defines 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 “Declaration Summary” on page 4-55, which serves as a summary and a reference. Refer to Table 4-1 for a complete list.

Table 4-1 Service Interface Data Types

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

Table 4-1 Service Interface Data Types

Data Type	Description
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 client
Real	Type definition for a Real data value[1]
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

1. This data type is implemented only for the Solstice CMIP product.

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

- **Sint**—positive and negative integers representable in 16 bits
- **Sint16**—positive and negative integers representable in 16 bits
- **Sint32**—positive and negative integers representable in 32 bits
- **Uint**—non-negative integers representable in 16 bits
- **Uint16**—non-negative integers representable in 16 bits
- **Uint32**—non-negative integers representable in 32 bits
- **Double**—positive and negative floating point numbers representable in 64 bits

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

C Declaration

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

4.1.1 Boolean

Description

Boolean is a data value of data type Boolean. 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.

C Declaration

```
typedef OM_uint32 OM_boolean;
```

4.1.2 Descriptor

Description

A *Descriptor* is a data value of type 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.

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.

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

Syntax (Syntax)
Identifies the syntax of the attribute value.

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

```
my_syntax = my_public_object[3].syntax & OM_S_SYNTAX
my_public_object[4].syntax =
    my_syntax + (my_public_object[4].syntax & ~OM_S_SYNTAX);
```

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

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

No-Value (Boolean)
Only true if the descriptor is service-generated and the value is not present (because exclude-values or exclude-multiples was set in the call to `om_get()`).

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

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

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

Service-Generated (Boolean)
It is true only if the descriptor is service-generated and the first descriptor of a public object, or the defined part of a private object. Refer to Chapter 5, “Workspace Interface”.

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

Private (Boolean)
It is true 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}`.

Value (Value)
The attribute value.

4.1.3 Enumeration

Description

Enumeration is a data value of type data that has an attribute value whose syntax is an enumeration syntax.

C Declaration

```
typedef OM_sint32 OM_enumeration;
```

4.1.4 Exclusions

Description

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

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

C Declaration

```
typedef OM_uint OM_exclusions;
```

4.1.5 Integer

Description

Integer is a data value that has an attribute value whose syntax is integer.

C Declaration

```
typedef OM_sint32 OM_integer;
```

4.1.6 Modification

Description

Modification is a data value that denotes a kind of modification, as defined by the `om_put()` function. It is chosen from the following set: **insert-at-beginning**, **insert-at-certain-point**, **insert-at-end**, **replace-all**, and **replace-certain-values**.

C Declaration

```
typedef OM_uint OM_modification;
```

4.1.7 Object

Description

Object is a data value that 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 ("Private Object" on page 4-11 and "Public Object" on page 4-12).

C Declaration

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

4.1.8 Object Identifier

Description

Object Identifier is a data value that contains an octet string which comprises the contents octets of the BER encoding of an ASN.1 object identifier.

C Declaration

```
typedef OM_string OM_object_identifier;
```

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 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 two macros `OM_IMPORT` and `OM_EXPORT`.

The `OM_IMPORT` macro makes the class or other object identifier constants available within a compilation unit.

```
OM_IMPORT(class_name)
OM_IMPORT(OID_name)
```

The `OM_EXPORT` macro allocates memory for the constants that represent the class or other object identifier.

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

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 “Declaration Summary” on page 4-55).

Use of Object Identifiers in C

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

This macro initializes a descriptor. It sets the type component to the syntax component to `{OM_S_OBJECT_IDENTIFIER_STRING}`, and value component to the object identifier.

```
OM_NULL_DESCRIPTOR
```

This macro initializes 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`, and the object identifier for BER rules has a variable called `OM_BER`. This variable can be supplied as an argument to functions when required. This variable is valid only when it is exported by an `OM_EXPORT` macro, and imported by an `OM_IMPORT` macro, in the compilation units that use it. This variable cannot form part of a descriptor, but the value of its length and elements components can be used.

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

#include <xom.h>

OM_IMPORT(OM_C_ENCODING)
OM_IMPORT(OM_CANONICAL_BER)

/*  The following sequence must appear in exactly one compilation
 *  unit in place of the above:
 *
 *  #include <xom.h>
 *
 *  OM_EXPORT(OM_C_ENCODING)
 *  OM_EXPORT(OM_CANONICAL_BER)
 */
main()
{
/* Use #1 - Define a public object of class Encoding
 *       (Note that xxxx is a Message Handling class which can be
 *           encoded.)
 */
OM_descriptor my_public_object[] = {
    OM_OID_DESC(OM_CLASS, OM_C_ENCODING),
    OM_OID_DESC(OM_OBJECT_CLASS, MA_C_xxxx),
    { OM_OBJECT_ENCODING, OM_S_ENCODING, some_BER_value },
    OM_OID_DESC(OM_RULES, OM_CANONICAL_BER),
    OM_NULL_DESCRIPTOR
    };

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

4.1.9 Private Object

Description

Private Object is a data value which is the handle for a private object. It comprises a single descriptor whose type component is private-object and whose syntax and value components are unspecified.

C Declaration

```
typedef OM_object OM_private_object;
```

Note – The descriptor's syntax and value components are essential to the service's proper operation with respect to the private object. However, the service-generated `{OM_S_SERVICE_GENERATED}` and private-object `{OM_S_PRIVATE}` bits in the syntax component are always set by the service.

4.1.10 Public Object

Description

Public Object is a data value which is a public object. It comprises one or (typically) more descriptors, all except 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 the values they represent. The order of the resulting groups of descriptors is unspecified.

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

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

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

C Declaration

```
typedef OM_object OM_public_object;
```

4.1.11 Real

Note – This data type is implemented only for the Solstice CMIP product.

Description

Real is a data value that has an attribute value whose syntax is real.

C Declaration

```
typedef OM_double OM_real;
```

4.1.12 Return Code

Description

Return Code is a data value which is the integer in the interval [0, 2^{16}), and denotes an outcome of an interface function. It is chosen from the set specified in “Functions” on page 4-20. All editions of this document employ integers in the narrower interval [0, 2^{15}) to denote the return codes they define.

C Declaration

```
typedef OM_uint OM_return_code;
```

4.1.13 String

Description

String is a data value string that is an instance of a string syntax. In the C declaration, a string is represented as either a length-specified or a null-terminated string. A string has the following components.

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) }
```

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

In the C declaration, 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.

Table 4-2 Elements of a String

	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 can supply a string value to the service in either length-specified or null-terminated form.

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

In the C declaration, the macro `{OM_STRING}`, is provided for constructing 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.

4.1.14 *Syntax*

Description

Syntax denotes an individual syntax, or a set of syntaxes taken together and is an integer data value in the interval [0, 2^9).

The data value is chosen from among the following:

- **boolean**, **integer** and **null**—each denotes the syntax of the same name
- **generalized-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**, **videotex-string** and **visible-string**—each denotes the string syntax of the same name

Integers are in the (narrower) interval [0, 2^9) and denote the syntaxes they define. The range [2^9, 2^{10}) is reserved for extensions by the implementor. 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.

C Declaration

```
typedef OM_uint16 OM_syntax;
```

4.1.15 *Type*

Description

Type is a data value that is an integer in the interval [0, 2^{16}). It denotes a type in the context of a package, except that the values no-more-types, and private-object have the meanings given them by the type list and private object data types, respectively. All editions of this document employ integers in the (narrower) interval [0, 2^{15}) to denote the types they define.

C Declaration

```
typedef OM_uint16 OM_type;
```

4.1.16 Type List

Description

Type List is a data value of an ordered sequence of zero or more type numbers, each an instance of the *type* data type. In the C interface, an additional data value, **no-more-types**, follows (and thus delimits) the sequence. The C representation of the sequence is an array.

C Declaration

```
typedef OM_type *OM_type_list;
```

4.1.17 Value

Description

Value is a data value that is an attribute value. It has no components if the value's syntax is **no-more-syntaxes**, or if the value's syntax is **no-value**, and (exactly) one of the following components.

C Declaration

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

Note – The structure above, in particular, its padding component, aligns the object component with the elements component of the string component below. This facilitates initialization in C.

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

Note – The identifier `OM_value_union` is defined for reasons of compilation order. It is used in the definition of the descriptor data type.

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

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

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

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

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

Note – 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 is present.

4.1.18 Value Length

Description

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

C Declaration

```
typedef OM_uint32 OM_value_length;
```

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

4.1.19 Value Position

Description

Value Position is a data value that is the integer in the interval [0, 2^{32}-1], which denotes the position of a value within an attribute, except that the value all-values has the meaning given to it by the `om_get()` function.

C Declaration

```
typedef OM_uint32 OM_value_position;
```

4.1.20 Workspace

Description

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

C Declaration

```
typedef void *OM_workspace;
```

Note – The workspace does not effect the client. This handle is for service implementors and is further explained in Chapter 5, “Workspace Interface”.

4.2 Functions

This section defines the functions of the service interface. The functions of both the generic and C interfaces are listed in Table 4-3. Those of the C interface are summarized in “Declaration Summary” on page 4-55.

Table 4-3 Service Interface Functions

Function	**Purpose**
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
OMPexamin	Examine the contents of an object.
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

The service interface comprises the functions listed in Table 4-3. The purpose and range of capabilities of these functions are summarized as follows:

- **Copy**—creates an independent copy of an existing private object, and all of its subobjects. The copy is placed in the original’s workspace, or in another specified by the client.
- **Copy Value**—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**—creates a new private object that is an instance of a particular class. The object may be initialized with the attribute values specified as initial in the definition of the class.

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

- **Decode**—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**—deletes a service-generated public object, or makes a private object inaccessible.
- **Encode**—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**—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**—determines whether an object is an instance of a particular class. The client can determine an object's class simply by inspection. The use 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.
- **OMPexamin**—examines the contents of an XOM object.
- **Put**—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**—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**—removes and discards particular values of an attribute of a private object. The attribute itself is removed if no values remain.
- **Write**—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 provided 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, 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.

4.2.1 om_copy()

Description

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

Synopsis

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

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

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 may be success or one of the values listed under Errors below.

Copy (Private Object)
This result is present if the copy result is successful.

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, too-many-values

4.2.2 om_copy_value()

Description

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

Synopsis

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

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

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 whose value 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 may be 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, wrong-value-type

4.2.3 om_create()

Description

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

Synopsis

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

OM_return_code
om_create (
    OM_object_identifier      class,
    OM_boolean                initialize,
    OM_workspace              workspace,
    OM_private_object         *object
);
```

Arguments

Class (Object Identifier)
Identifies the class of the object to be created. The specified class must be concrete.

Initialize (Boolean)
Whether the created object is to be initialized 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 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 possible 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 may be success or one of the values listed under Errors below.

Object (Private Object)
The created object. This result is present 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, temporary-error

4.2.4 om_decode()

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.

Synopsis

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

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

Arguments

Encoding (Private Object)
The remaining accessible encoding. It shall be an instance of class encoding.

Results

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be success or one of the values listed under Errors below.

Original (Private Object)
The original, which is created in the encoding's workspace. This result is present if, and only if, the Return Code result is success.

Errors

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

4.2.5 om_delete()

Description

This function deletes a service-generated public object, or makes a private object inaccessible. It is not intended for use on client-generated public objects. If applied to a service-generated public object, the function deletes the object and releases any resources associated with the object, including the space occupied by descriptors and attribute values. The function is applied recursively to any public subobjects. There is no effect on any private subobjects. The user should not use `om_delete()` directly on public subobjects (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.

Synopsis

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

OM_return_code
om_delete (
    OM_object      subject
);
```

Arguments

Subject (Object)
The object to be deleted.

Results

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be 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, temporary-error

4.2.6 om_encode()

Description

This function creates a new private object, which encodes an existing private object. The client identifies the set of rules that the service is to follow to produce the new object, known as 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. The identities of these latter classes are implementation-defined.

Synopsis

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

OM_return_code
om_encode (
    OM_private_object     original,
    OM_object_identifier  rules,
    OM_private_object     *encoding
);
```

Arguments

Original (Private Object)
The original that 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 may be 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, temporary-error

4.2.7 om_get()

Description

This function creates a public copy of all or particular parts of a private object. The client may request certain exclusions, each of which reduces the copy to a portion of the original. One exclusion is always requested implicitly. For each attribute value in the original that is a string whose length exceeds an implementation-defined number, the copy includes a descriptor that omits the elements (but not the length) of the string; the elements component of the String component of the Value component of the descriptor is **elements-unspecified**, and the **Long-String** bit of the Syntax component is set to **true**.

Note – The client can access long values using the Read function.

Synopsis

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

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

Arguments

Original (Private Object)
The original that 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 earlier exclusions having precedence over later exclusions. If, after the application of an exclusion, a portion of the object would not be returned, no further exclusions need be applied to that portion.

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

   This exclusion provides a means for determining the values of specific attributes, as well as the syntaxes of those values.

2. **exclude-multiples**—the copy includes a single descriptor for each attribute 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 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.

3. **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.

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

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

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

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

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

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

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

Initial Value (Value Position)
Present if the **exclude-all-but-these-values** exclusion is requested, the position within each attribute of the first value to be included in the copy.

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

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

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

Results

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be success or one of the values listed under Errors below.

Copy (Public Object)
This result is present if, and only if, the Return Code result is a 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, wrong-value-syntax, wrong-value-type

4.2.8 om_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 `om_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.

Synopsis

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

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

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 may be 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, temporary-error

4.2.9 om_put()

Description

This function places or replaces in one private object, the destination, copies of the attribute values of another object, the source. The source can be a public or private object.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).

Synopsis

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

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

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 `om_put()` function uses the attribute values in the source to modify the destination. 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 the following:

- **insert-at-beginning**—the source values are inserted before any existing destination values. The destination values are retained.

- **insert-at-certain-point**—the source values are inserted before the value at a specified position in the destination attribute. The destination values are retained.
- **insert-at-end**—the source values are inserted after any existing destination values. The destination values are retained.
- **replace-all**—the source values are placed in the destination attribute. The existing destination values, if any, are discarded. The destination values are discarded.
- **replace-certain-values**—the source values are substituted for the values at specified positions in the destination attribute. The destination values are discarded.

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

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

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

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

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

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

Results

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be 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-adjacen, wrong-value-length, wrong-value-makeup, wrong-value-number, wrong-value-position, wrong-value-syntax, wrong-value-type

4.2.10 om_read()

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. The `om_read()` function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.

Synopsis

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

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

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 specific 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 realized 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 may be 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 realized 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, wrong-value-syntax

4.2.11 om_remove()

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. If the value is a subobject, the value is first removed and then the `om_delete()` function is applied to it, thus destroying the object.

Synopsis

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

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

Arguments

Subject (Private Object)
The subject which remains accessible. The subject's class is unchanged.

Type (Type)
Identifies the type of the attribute, some of whose values are to be removed. The type must not be Class.

Initial Value (Value Position)
The position within the 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 and all data values are removed.

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 and all data values are removed.

Results

Return Code (Return Code)
Whether the function succeeded and, if not, why. It may be 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, temporary-error

4.2.12 om_write()

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.

Synopsis

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

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

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

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

Starting Position (String Offset)
The offset, in octets, of the start of the string segment to write. If it exceeds the current length of the string value being written, the argument is taken to be equal to that current length. In the C interface, the Starting Position argument and the Next Position result of the generic interface are realized 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 may be 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 realized 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, wrong-value-syntax

4.2.13 OMPexamin()

Description

This function displays the content of an object.

Synopsis

```
#include <xom.h>

void
OMPexamin (
    OM_object      object,
    int            level,
    int            mode
);
```

Arguments

Object
This is the object that will be displayed.

Level
This is the object’s indentation level. For instance, the output can be shifted to the right if the level is not 0.

Mode

The mode of display can be stipulated with 1 or 2. 1 is the *ugly* display mode and 2 is the *pretty* display mode.

For example, a display of an object in ugly mode is:

```
[02] PRIV - Top=1 Rnb=0 Ubs=0
[V2] T=0003  S=OidStr L=10 V(mseg)=0x2a863a00881a0601880b
[V2] T=11033 S=Int    V=0
[V2] T=11057 S=Obj    V->
     [03] PRIV - Top=0 Rnb=0 Ubs=0
     [V3] T=0003  S=OidStr L=10 V(mseg)=0x2a863a00881a06018805
     [V3] T=11043 S=OctStr L=7  V(mseg)=0x6d656c6f646965
     [V3] T=11053 S=OctStr L=4  V(mseg)=0x74703430
     [V3] T=11067 S=OctStr L=3  V(mseg)=0x707273
     [V3] T=11069 S=OctStr L=3  V(mseg)=0x736573
```

For example, a display of the same object using pretty mode would be:

```
    Class=Session
    FileDescriptor=[Int]=0
    RequestorAddress=(PRIV subobject)
        Class=PresentationAddress
        NAddresses=[OctStr]=0x6d656c6f646965
        PSelector=[OctStr]=0x74703430
        SSelector=[OctStr]=0x507273
        TSelector=[OctStr]=0x736573
```

4.3 Return Codes

This section defines 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 “Declaration Summary” on page 4-55.

Table 4-4 lists in the first column the return codes. The other columns identify with an “x” the return codes that apply to each function.

Table 4-4 Service Interface Return Codes

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
encoding-invalid	-	-	-	x	-	-	-	-	-	-	-	-
function-declined	-	x	x	-	-	x	-	-	x	-	x	x
function-interrupted	x	x	x	x	x	x	x	x	x	x	x	x
memory-insufficient	x	x	x	x	x	x	x	x	x	x	x	x
network-error	x	x	x	x	x	x	x	x	x	x	x	x
no-such-class	x	-	x	x	-	-	-	x	x	-	-	-
no-such-exclusion	-	-	-	-	-	-	x	-	-	-	-	-
no-such-modification	-	-	-	-	-	-	-	-	x	-	-	-
no-such-object	x	x	-	x	x	x	x	x	x	x	x	x
no-such-rules	-	-	-	x	-	x	-	-	-	-	-	-
no-such-syntax	-	-	-	-	x	-	-	x	x	-	-	x
no-such-type	-	x	-	-	x	-	x	-	x	x	x	x
no-such-workspace	x	-	x	-	-	-	-	-	-	-	-	-
not-an-encoding	-	-	-	x	-	-	-	-	-	-	-	-
not-concrete	-	-	x	-	-	-	-	-	x	-	-	-
not-present	-	x	-	-	-	-	-	-	x	x	-	x
not-private	x	x	-	x	-	x	x	-	x	x	x	x
not-the-services	-	-	-	-	x	-	-	x	-	-	-	-
permanent-error	x	x	x	x	x	x	x	x	x	x	x	x
pointer-invalid	x	x	x	x	x	x	x	x	x	x	x	x
success	x	x	x	x	x	x	x	x	x	x	x	x
system-error	x	x	x	x	x	x	x	x	x	x	x	x
temporary-error	x	x	x	x	x	x	x	x	x	x	x	x

Table 4-4 Service Interface Return Codes

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
too-many-values	x	-	-	x	-	-	-	-	x	-	-	-
values-not-adjacent	-	-	-	-	-	-	-	-	x	-	-	-
wrong-value-length	-	x	-	x	-	-	-	-	x	-	-	x
wrong-value-makeup	-	-	-	x	-	-	-	-	x	-	-	x
wrong-value-number	-	-	-	x	-	-	-	-	x	-	-	-
wrong-value-position	-	-	-	-	-	-	-	-	x	-	-	x
wrong-value-syntax	-	x	-	x	-	-	x	-	x	x	-	x
wrong-value-type	-	x	-	x	-	-	x	-	x	-	-	-

The return codes are as follows:

- **success**—the function completed successfully.
- **encoding-invalid**—the octets that constitute the value of an encoding's Object Encoding attribute are invalid.
- **function-declined**—the function does not apply to the object to which it is addressed.
- **function-interrupted**—the function was aborted by an external force (e.g., a keystroke, designated for this purpose, at a user interface).
- **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.

4.4 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 “Data Types” on page 4-1 and “Functions” on page 4-20.

Reference to the function macros that appear in “Types and Macros” on page 5-4, the specification of the workspace interface here replace the function prototypes that appear (in “Functions” on page 4-20) in the specification of the service interface.

The declarations are included in header file `xom.h`. The header file includes by reference a second header file `xomi.h` comprising the declarations defining the C workspace interface (see Chapter 5, “Workspace Interface”). The symbols the declarations defined 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_lengthlength;
   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_uint32padding;
    OM_objectobject;
} OM_padded_object;

typedef union OM_value_union {
    OM_string  string;
    OM_boolean boolean;
    OM_enumerationenumeration;
    OM_integer integer;
    OM_padded_objectobject;
} OM_value;
```

```
/* Value Length */
typedef OM_uint32 OM_value_length;

/* Value Position */
typedef OM_uint32 OM_value_position;

/* TERTIARY DATA TYPES */

/* Descriptor */
typedef struct OM_descriptor_struct {
   OM_type           type;
   OM_syntax      syntax;
   union OM_value_unionvalue;
} 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 "\x56\x06\x01\x02\x04"

/* Constant for the Encoding class */

#define OMP_O_OM_C_ENCODING "\x56\x06\x01\x02\x04\x01"

/* Constant for the External class */

#define OMP_O_OM_C_EXTERNAL "\x56\x06\x01\x02\x04\x02"
```

```
/* Constant for the Object class */

#define OMP_O_OM_C_OBJECT "\x56\x06\x01\x02\x04\x03"

/* Constant for the BER Object Identifier */

#define OMP_O_OM_BER "\x51\x01"

/* Constant for the Canonical-BER Object Identifier */

#define OMP_O_OM_CANONICAL_BER "\x56\x06\x01\x02\x04\x04"

/* Return Code */

#define OM_SUCCES ( (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_UTC_TIME_STRING ( (OM_syntax) 23 )
#define OM_S_VIDEOTEX_STRING ( (OM_syntax) 21 )
#define OM_S_VISIBLE_STRING ( (OM_syntax) 26 )
#define OM_S_LONG_STRING ((OM_syntax) 0x8000)
#define OM_S_NO_VALUE ((OM_syntax) 0x4000)
#define OM_S_LOCAL_STRING ((OM_syntax) 0x2000)
#define OM_S_SERVICE_GENERATED ((OM_syntax) 0x1000)
#define OM_S_PRIVATE ((OM_syntax) 0x0800)
#define OM_S_SYNTAX ((OM_syntax) 0x03FF)
```

```
/* Type */

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

/* Value Position */

#define OM_ALL_VALUES ( (OM_value_position) 0xFFFFFFFF )

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

/* END SERVICE INTERFACE */
```

Workspace Interface 5

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.

- 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. We refer to these as the *-1st* and *0th* elements.

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

Note – 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:

- Inspect the *type* component. If it is `{OM_PRIVATE_OBJECT}`, the object is a PRI. Otherwise, it is a SPUB or CPUB.
- 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.

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

Note also that if a Service-Generated Public Object has public subobjects (for example, due to an `om_get` with no exclusions), the -1st descriptor will exist and the `OM_S_SERVICE_GENERATED` bit will be set on all public subobjects.

5.3 Types and Macros

5.3.1 Standard Internal Representation of an Object

The `OMP_object_header` and `OMP_object` types provide the standard representation.

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

The {`OM_S_SERVICE_GENERATED`} bit in the Syntax component of a descriptor of Type {`OM_CLASS`} determines whether a service allocated the storage for the descriptor array.

5.3.2 Standard Internal Representation of a Workspace

```
typedef OM_return_code
(*OMP_copy) (
    OM_private_object         original,
    OM_workspace              workspace,
    OM_private_object         *copy
);
typedef OM_return_code
(*OMP_copy_value) (
    OM_private_object         source,
    OM_type                   source_type,
    OM_value_position         source_value_position,
    OM_private_object         destination,
    OM_type                   destination_type,
    OM_value_position         destination_value_position
);
typedef OM_return_code
(*OMP_create) (
    OM_object_identifier      class,
    OM_boolean                initialise,
    OM_workspace              workspace,
    OM_private_object         *object
);
```

```
typedef OM_return_code
(*OMP_decode) (
    OM_private_object      encoding,
    OM_private_object      *original
);
typedef OM_return_code
(*OMP_delete) (
    OM_object              subject
);
typedef OM_return_code
(*OMP_encode) (
    OM_private_object      original,
    OM_object_identifier   rules,
    OM_private_object      *encoding
);
typedef OM_return_code
(*OMP_get) (
    OM_private_object      original,
    OM_exclusions          exclusions,
    OM_type_list           included_types,
    OM_boolean             local_strings,
    OM_value_position      initial_value,
    OM_value_position      limiting_value,
    OM_public_object       *copy,
    OM_value_position      *total_number
);
typedef OM_return_code
(*OMP_instance) (
    OM_object              subject,
    OM_object_identifier   class,
    OM_boolean             *instance
);
typedef OM_return_code
(*OMP_put) (
    OM_private_object      destination,
    OM_modification        modification,
    OM_object              source,
    OM_type_list           included_types,
    OM_value_position      initial_value,
    OM_value_position      limiting_value
);
```

```
typedef OM_return_code
(*OMP_read) (
    OM_private_object      subject,
    OM_type                type,
    OM_value_position      value_position,
    OM_boolean             local_string,
    OM_string_length       *string_offset,
    OM_string              *elements
);
typedef OM_return_code
(*OMP_remove) (
    OM_private_object      subject,
    OM_type                type,
    OM_value_position      initial_value,
    OM_value_position      limiting_value
);
typedef OM_return_code
(*OMP_write) (
    OM_private_object      subject,
    OM_type                type,
    OM_value_position      value_position,
    OM_syntax              syntax,
    OM_string_length       *String_offset,
    OM_string              elements
);
typedef struct OMP_functions_body {
    OM_uint32              function_number;
    OMP_copy               omp_copy;
    OMP_copy_value         omp_copy_value;
    OMP_create             omp_create;
    OMP_decode             omp_decode;
    OMP_delete             omp_delete;
    OMP_encode             omp_encode;
    OMP_get                omp_get;
    OMP_instance           omp_instance;
    OMP_put                omp_put;
    OMP_read               omp_read;
    OMP_remove             omp_remove;
    OMP_write              omp_write;
} OMP_functions;

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

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.

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.

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

5.3.3 Useful Macros

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

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

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

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

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

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

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

```
#define OMP_WORKSPACE(external)\

((OMP_workspace)(OMP_INTERNAL(external)>value.string.elements))
```

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

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

5.4 Dispatcher Macros

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

```
#define om_copy(ORIGINAL,WORKSPACE,COPY)\
    (((OMP_workspace)(WORKSPACE))->functions->omp_copy(\
                 (ORIGINAL),(WORKSPACE),(COPY)))
#define om_copy_value(SOURCE, SOURCE_TYPE, SOURCE_POSITION,\
                       DEST,   DEST_TYPE,   DEST_POSITION)\
        (OMP_FUNCTIONS(DEST)->omp_copy_value(\
                       (SOURCE), (SOURCE_TYPE),
(SOURCE_POSITION),\
                       (DEST),   (DEST_TYPE),   (DEST_POSITION)))
#define om_create(CLASS,INITIALISE,WORKSPACE,OBJECT)\
     (((OMP_workspace)(WORKSPACE))->functions->omp_create(\
                   (CLASS),(INITIALISE),(WORKSPACE),(OBJECT)))
#define om_decode(ENCODING,ORIGINAL)\
    (OMP_FUNCTIONS(ENCODING)->omp_decode((ENCODING),(ORIGINAL)))
#define om_delete(SUBJECT)\
    (((SUBJECT)->syntax & OM_S_SERVICE_GENERATED) ?\
    OMP_FUNCTIONS(SUBJECT)->omp_delete((SUBJECT)) :\
    OM_NOT_THE_SERVICES)
```

```
#define om_encode(ORIGINAL,RULES,ENCODING)\
        (OMP_FUNCTIONS(ORIGINAL)-
>omp_encode((ORIGINAL),(RULES),(ENCODING)))
#define om_get(ORIGINAL,EXCLUSIONS,TYPES,LOCAL_STRINGS,\
           INITIAL,LIMIT,COPY,TOTAL_NUMBER)\
        (OMP_FUNCTIONS(ORIGINAL)->omp_get(\
         (ORIGINAL),(EXCLUSIONS),(TYPES),(LOCAL_STRINGS),\
           (INITIAL),(LIMIT),(COPY),(TOTAL_NUMBER)))
#define om_instance(SUBJECT,CLASS,INSTANCE)\
    (((SUBJECT)->syntax & OM_S_SERVICE_GENERATED) ?\
    OMP_FUNCTIONS(SUBJECT)-
>omp_instance((SUBJECT),(CLASS),(INSTANCE)) :\
    OM_NOT_THE_SERVICES)
#define
om_put(DESTINATION,MODIFICATION,SOURCE,TYPES,INITIAL,LIMIT)\
        (OMP_FUNCTIONS(DESTINATION)->omp_put(\

(DESTINATION),(MODIFICATION),(SOURCE),(TYPES),(INITIAL),(LIMIT)
))
#define om_read(SUBJECT,TYPE,VALUE_POS,LOCAL_STRING,\
           STRING_OFFSET,ELEMENTS)\
        (OMP_FUNCTIONS(SUBJECT)->omp_read(\
             (SUBJECT),(TYPE),(VALUE_POS),(LOCAL_STRING),\
       (STRING_OFFSET),(ELEMENTS)))
#define om_remove(SUBJECT,TYPE,INITIAL,LIMIT)\
        (OMP_FUNCTIONS(SUBJECT)->omp_remove(\
                 (SUBJECT),(TYPE),(INITIAL),(LIMIT)))
#define
om_write(SUBJECT,TYPE,VALUE_POS,SYNTAX,STRING_OFFSET,ELEMENTS)\
        (OMP_FUNCTIONS(SUBJECT)->omp_write(\
             (SUBJECT),(TYPE),(VALUE_POS),\
             (SYNTAX),(STRING_OFFSET),(ELEMENTS)))
```

Object Management Package 6

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

6.1 Class Hierarchy

This section shows the hierarchical organization of the OM classes. The names of abstract classes are in italics. In the example below, 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**

6.2 Class Definitions

This section defines the OM classes.

6.2.1 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 (example, a directory or message transfer system) an object it does not recognize.

This class has the attributes of its superclass (*Object*) and the specific attributes listed in Table 6-1.

Table 6-1 Encoding Attributes

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

[1] If the Rules attribute is **ber** or **canonical-ber**, the syntax of the present attribute shall be String (Encoding).

Object-Class
Identifies the class of the object that the Object Encoding attribute encodes. The class must be concrete.

Object-Encoding
The encoding itself.

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

- **ber**. Specified in ASN.1 as *{joint-iso-ccitt asn1(1) basic-encoding(1)}*, this value denotes the BER (see Clause 25.2 of *Recommendation X.208*, Appendix A, “Referenced Documents”).

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

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.

6.2.2 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 Appendix A, “Referenced Documents”).

This class has the attributes of its superclass (Object) and the specific attributes listed in Table 6-2.

Table 6-2 Attributes Specific to External

Attribute	Value Syntax	Value Length	Value Number	Value Initially
Arbitrary Encoding	String (Bit)	-	0-1[1]	-
ASN.1 Encoding	String (Encoding)	-	0-1[1]	-
Data Value Descriptor	String (Object Descriptor)	-	0-1	-
Direct Reference	String (Object Identifier)	-	0-1[2]	-
Indirect Reference	Integer	-	0-1[2]	-
Octet Aligned Encoding	String (Octet)	-	0-1[1]	-

[1] Exactly one of these three attributes shall be present.
[2] At least one of these two attributes shall be present.

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 Appendix A, “Referenced Documents”).

ASN1 Encoding

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

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

Data Value Descriptor

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

Direct Reference

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

Indirect Reference

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

Octet Aligned Encoding

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

6.2.3 Object

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

The attributes specific to this class are listed in Table 6-3.

Table 6-3 Attributes Specific to Object

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

Class
Identifies the object's class.

Referenced Documents A

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

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

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

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

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

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

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

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

Glossary

Abstract Class	A class, instances of which are forbidden.
Accessible Object	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 standardized 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 Class

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, having no superclasses that are themselves subclasses of that class.

Immediate Subobject

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

Immediate Superclass

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

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 realize 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 Application Program Interface	The interface this document defines. Now known as OSI-Abstract-Data Manipulation (XOM) API.
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 realized, for the client's benefit, by the service as a whole.
Service	Software that implements the interface.
Specific	The attribute types, with respect to a class, 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 realized, 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