X/Open Guide

Guide to Selected X.400 and Directory Services APIs

X/Open Company, Ltd.

X/Open Guide

Guide to Selected X.400 and Directory Services APIs

ISBN: 1 872630 23 5
X/Open Document Number: G150

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

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

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

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

GUIDE TO SELECTED X.400 AND DIRECTORY SERVICES APIs

Preface

X.400 API ASSOCIATION BACKGROUND

To promote X.400 as the industry standard electronic mail service among the many existing and incompatible proprietary solutions, the X.400 API Association (XAPIA) was founded in January 1989.

The XAPIA's objective was to allow the widely installed base of non-X.400 electronic mail systems to exchange mail among themselves and with the international network of native X.400 users with minimal effort and cost in adding X.400 to existing mail systems by dividing the work among manufacturers.

The strategy for achieving this objective is to develop and promote standard *Application Program Interfaces* (APIs). An API can be regarded as the set of rules by which two pieces of software residing on the same computer system communicate through an operating system or a programming language. This contrasts with the definition of an Application Protocol (e.g., X.400 P1) as the set of rules by which two different computer systems communicate through a network.

In June 1989, the XAPIA with collaborative effort from more than 20 companies published the **X.400 Gateway API Specification**. Subsequently, X/Open, a consortium of major vendors developing a POSIX-compliant common application environment, considered the object-oriented approach adopted by XAPIA suitable for an X.500 API and decided to collaborate.

Consequently, in September 1990, XAPIA and X/Open jointly published Preliminary Specifications for a set of new APIs:

- the API to Electronic Mail (X.400) Preliminary Specification, which encompasses both the previous X.400 Gateway API and X.400 Application API (XMHS - see **Referenced Documents**).
- the X.500 API to Directory Services (XDS) Preliminary Specification - see **Referenced Documents**).
- the API to OSI Object Management (XOM) Preliminary Specification - see **Referenced Documents**).

These are expected to progress towards publication as full CAE (Common Applications Environment) Specifications before the end of 1991.

X/Open

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

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

The components of the Common Applications Environment are defined in X/Open CAE Specifications. These contain, among other things, an evolving portfolio of practical application programming interfaces (APIs), which significantly enhance portability of application programs at the source code level, and definitions of, and references to, protocols and protocol profiles, which significantly enhance the interoperability of applications.

The X/Open CAE Specifications are supported by an extensive set of conformance tests and a distinct X/Open trade mark - the XPG brand - that is licensed by X/Open and may be carried only on products that comply with the X/Open CAE Specifications.

The XPG brand, when associated with a vendor's product, communicates clearly and unambiguously to a procurer that the software bearing the brand correctly implements the corresponding X/Open CAE Specifications. Users specifying XPG conformance in their procurements are therefore certain that the branded products they buy conform to the CAE Specifications.

X/Open is primarily concerned with the selection and adoption of standards. The policy is to use formal approved *de jure* standards, where they exist, and to adopt widely supported *de facto* standards in other cases.

Where formal standards do not exist, it is X/Open policy to work closely with standards development organisations to assist in the creation of formal standards covering the needed functions, and to make its own work freely available to such organisations. Additionally, X/Open has a commitment to align its definitions with formal approved standards.

X/Open Specifications

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the long-life specifications that form the basis for conformant and branded X/Open systems. They are intended to be used widely within the industry for product development and procurement purposes.

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

CAE Specifications are not published to coincide with the launch of a particular XPG brand, but are published as soon as they are developed. By providing access to its specifications in this way, X/Open makes it possible for products that conform to the CAE (and hence are eligible for a future XPG brand) to be developed as soon as practicable, enhancing the value of the XPG brand as a procurement aid to users.

- *Preliminary Specifications*

  These are specifications, usually addressing an emerging area of technology, and consequently not yet supported by a base of conformant product implementations, that are released in a controlled manner for the purpose of validation through practical implementation or prototyping. A Preliminary Specification is not a ‘‘draft’’ specification. Indeed, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE Specification.

  Preliminary Specifications are analogous with the ‘‘trial-use’’ standards issued by formal standards organisations, and product development teams are intended to develop products on the basis of them. However, because of the nature of the technology that a Preliminary Specification is addressing, it is untried in practice and may therefore change before being published as a CAE Specification. In such a case the CAE Specification will be made as upwards-compatible as possible with the corresponding Preliminary Specification, but complete upwards-compatibility in all cases is not guaranteed.

In addition, X/Open periodically publishes:

- *Snapshots*

  Snapshots are ‘‘draft’’ documents, which provide a mechanism for X/Open to disseminate information on its current direction and thinking to an interested audience, in advance of formal publication, with a view to soliciting feedback and comment.

  A Snapshot represents the interim results of an X/Open technical activity. Although at the time of publication X/Open intends to progress the activity towards publication of an X/Open Preliminary or CAE Specification, X/Open is a consensus organisation, and makes no commitment regarding publication.

  Similarly, a Snapshot does not represent any commitment by any X/Open member to make any specific products available.

X/Open Guides

X/Open Guides provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant.

X/Open Guides are not normative, and should not be referenced for purposes of specifying or claiming X/Open-conformance.

This Document

This document is a Guide (see above). It is intended to provide an introduction and tutorial that complements the following specifications jointly published by XAPIA and X/Open:

- API to Electronic Mail (X.400) CAE Specification
- API to Directory Services CAE Specification
- API to OSI Object Management CAE Specification

The tutorial is illustrated through the use of selected 'C' language programming examples.

While readers are not expected to have in-depth knowledge of the CCITT X.400 and X.500 Recommendations and the above specifications from XAPIA and X/Open, they are assumed to have some knowledge of these specifications. Extensive reference is made to these documents throughout this tutorial. Readers should also be familiar with the programming language 'C', so as to follow the programming examples given: note that compatibility of the programming examples with ISO C has not been verified.

Trademarks

Referenced Documents

EDI	Draft Recommendation X.435, ‘‘Message Handling Systems: EDI Messaging System’’ Version 5.0, International Telegraph and Telephone Consultative Committee (CCITT), 1990.
LIB	ISO CD 10160, ‘‘Documentation - Interlibrary Loan Service Definition’’.
Manros	Carl-Uno Manros, The X.400 Blue Book Companion, Technology Appraisals, 1989.
MHS-1984	CCITT X.400 (1984) includes the following: — Recommendation X.400, ‘‘Message Handling Systems: System Model - Service Elements’’, International Telegraph and Telephone Consultative Committee (CCITT) Red Book, Fascicle VIII.7, International Telecommunications Union, 1984, pp. 3-38. — Recommendation X.401, ‘‘Message Handling Systems: Basic Service Elements and Optional User Facilities’’, Ibid., pp. 39-45. — Recommendation X.408, ‘‘Message Handling Systems: Encoded Information Type Conversion Rules’’, Ibid., pp. 46-61. — Recommendation X.409, ‘‘Message Handling Systems: Presentation Transfer Syntax and Notation’’, Ibid., pp. 62-93. — Recommendation X.410, ‘‘Message Handling Systems: Remote Operations and Reliable Transfer Service’’, Ibid., pp. 93-126. — Recommendation X.411, ‘‘Message Handling Systems: Message Transfer Layer’’, Ibid., pp. 127-182. — Recommendation X.420, ‘‘Message Handling Systems: Interpersonal Messaging User Agent Layer’’, Ibid. pp 182-219. — Recommendation X.430, ‘‘Message Handling Systems: Access Protocol for Teletex Terminals’’, Ibid. pp 219-266.
MHS-1988	CCITT X.400 (1988) includes the following: — Recommendation X.400, ‘‘Message Handling Systems: System Model - Service Elements’’, International Telegraph and Telephone Consultative Committee (CCITT) Blue Book, Fascicle VIII.7, International Telecommunications Union, 1988. See also ISO 10021-1. — Recommendation X.402, ‘‘Message Handling Systems: Overall Architecture’’, Ibid. See also ISO 10021-2. — Recommendation X.403, ‘‘Message Handling Systems: Conformance Testing’’, Ibid. — Recommendation X.407, ‘‘Message Handling Systems: Abstract Service Definition Conventions’’, Ibid. See also ISO 10021-3.

- Recommendation X.408, ''Message Handling Systems: Encoded Information Type Conversion Rules'', Ibid.
- Recommendation X.411, ''Message Handling Systems: Message Transfer System: Abstract Service Definition and Procedures'', Ibid. See also ISO 10021-4.
- Recommendation X.413, ''Message Handling Systems: Message Store: Abstract Service Definition'', Ibid. See also ISO 10021-5.
- Recommendation X.419, ''Message Handling Systems: Protocol Specifications'', Ibid. See also ISO 10021-6.
- Recommendation X.420, ''Message Handling Systems: Interpersonal Messaging System'', Ibid. See also ISO 10021-7.

Radicati	Sara Radicati, Electronic Mail: An Introduction to the X.400 Message Handling Standards, McGraw-Hill.
Steedman	Douglas Steedman, ASN.1 the Tutorial and Reference, Technology Appraisals, 1990.
SWIFT	Society for Worldwide International Financial Telecommunications, IFT User Handbook, Version 1.0.
X.208	Recommendation X.208, ''Specification of Abstract Syntax Notation One (ASN.1)'', International Telegraph and Telephone Consultative Committee (CCITT), International Telecommunications Union, 1988. See also ISO 8824.
X.209	Recommendation X.209, ''Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)'', International Telegraph and Telephone Consultative Committee (CCITT), International Telecommunications Union, 1988. See also ISO 8825.
X.500	X.500 series includes the following:

- Recommendation X.500, ''The Directory - Overview of Concepts, Models and Service'', International Telegraph and Telephone Consultative Committee (CCITT), International Telecommunications Union, 1988. See also ISO 9594-1.
- Recommendation X.501, ''The Directory -- Models'', Ibid. See also ISO 9594-2.
- Recommendation X.509, ''The Directory - Authentication Framework'', Ibid. See also ISO 9594-8.
- Recommendation X.511, ''The Directory - Abstract Service Definition'', Ibid. See also ISO 9594-3.
- Recommendation X.518, ''The Directory - Procedures for Distributed Operation'', Ibid. See also ISO 9594-4.
- Recommendation X.519, ''The Directory - Protocol Specifications'', Ibid. See also ISO 9594-5.

	— Recommendation X.520, ‘‘The Directory - Selected Attribute Types’’, Ibid. See also ISO 9594-6.
	— Recommendation X.521, ‘‘The Directory - Selected Object Classes’’, Ibid. See also ISO 9594-7.
XDS	X.400 API Association and X/Open Company Limited, X/Open API to Directory Services CAE Specification.
XEDI	X.400 API Association and X/Open Company Limited, EDI Messaging Package Specification.
XMHS	X.400 API Association and X/Open Company Limited, API to Electronic Mail (X.400) CAE Specification.
XMS	X.400 API Association and X/Open Company Limited, X.400 Message Store Specification.
XOM	X.400 API Association and X/Open Company Limited, OSI-Abstract-Data Manipulation API (XOM) CAE Specification.

Chapter 1

Introduction

1.1 MESSAGE HANDLING STANDARDS OVERVIEW

The CCITT X.400-series of Recommendations were first approved in 1984. Broadly speaking, they define a general-purpose *Message Transfer System* (MTS) with its own communications system (based on Open System Interconnection (OSI)), its own addressing scheme (OR-Name) and its own syntax (X.409 and later ASN.1 (MHS-1984:X.409, X.208, X.209 - see **Referenced Documents**)).

The Recommendations also specify one particular application of the MTS, the *Interpersonal Messaging* (IM) Service. IM is a service that deals with a message format which looks much like a standard office memo - the kind most people associate with ‘‘electronic mail’’.

Some functional objects were defined to implement these functionalities.

- The *User Agent* (UA) is the interface to the user (which can be human or a computer program). The UA will allow the user to generate a message and will also render incoming messages to the user.
- The *Message Transfer Agent* (MTA) is responsible for routing messages within the MTS. An MTA will permit properly authorised UAs to submit messages to one or more recipients and will deliver messages to UAs. An MTA can also transfer messages to other MTAs that will, in turn, take further action as required; i.e., either transfer the message, deliver it or, if delivery or transfer is not possible, discard it, returning an advice called a *non-delivery report* to the originator. Refer to **Figure 1-1**.

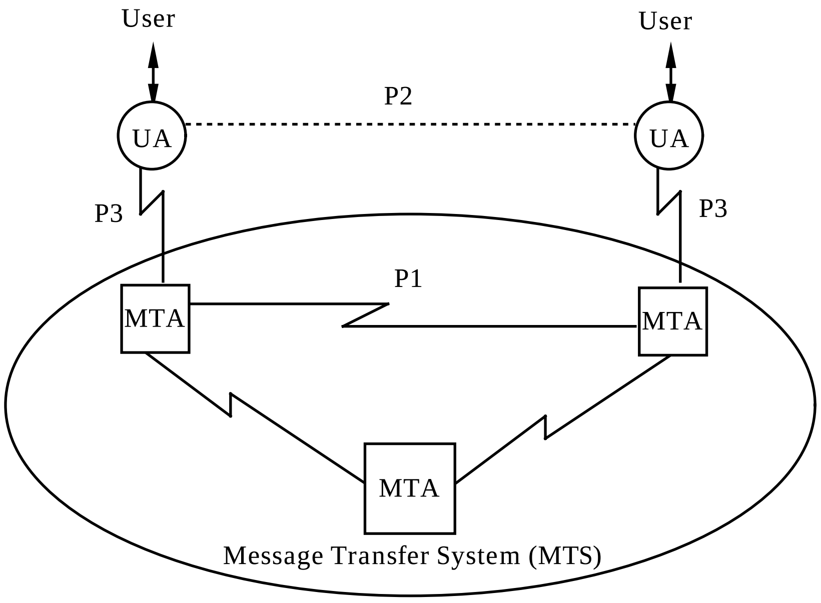


Figure **1-1.**

Some ‘‘protocols’’ were defined to facilitate communication among the functional objects. P1 is used between MTAs as the transfer protocol; it is often referred to as an ‘‘envelope’’; P2 is the content for Interpersonal Messages, and P3 is used between a UA and its MTA for both submissions and deliveries.

In 1988, the CCITT approved a substantial set of additions to the X.400 series. Among these additions are:

- extensive security features
- a Message Store and a new protocol P7
- Distribution Lists
- use of the OSI Directory (X.500)
- several additions to Message Transfer Services and its associated protocol P1
- some minor additions to Interpersonal Messaging Services and its associated protocol P2
- methods for extending X.400
- a revised OSI communications system

Most of these items are outside the scope of this document. Interested readers are referred to Radicati, MHS-1988 and Manros (see **Referenced Documents**) for more information. However, the *Message Store* (MS) is of interest. MS is a new functional object inserted between the UA and the MTA. It stores delivered messages until the UA wants to retrieve them. A new protocol was defined for this purpose: P7. As shown in **Figure 1-2**, the MS can be separated from the MTA (i.e., located on a different computer system) in which case the MS-to-MTA protocol is P3. The XAPIA and X/Open also provide an API specification for the MS (refer to XMS - see **Referenced Documents**).

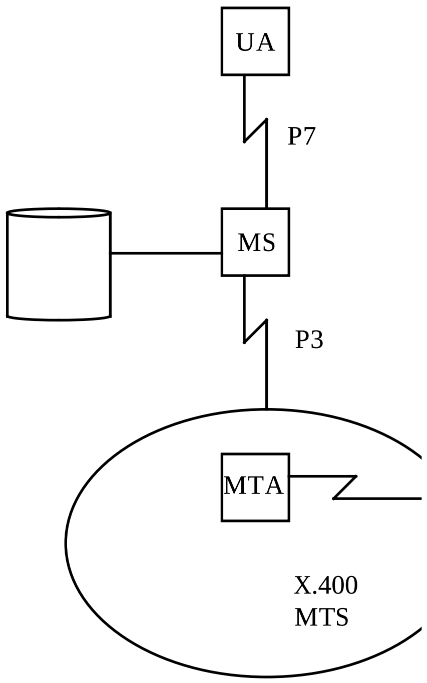


Figure **1-2.**

ISO/IEC collaborated with CCITT in the 1985-1988 study period and has published its own set of standards (ISO 10021 parts 1-7) referred to as **MOTIS (Message Oriented Text Interchange System)**. The CCITT and ISO documents are generally identical except for the following key points:

- ISO 10021 omitted the equivalent of X.403 (Conformance Testing).
- ISO 10021 omitted the equivalent of X.408 (EIT conversion).
- ISO has added two information base types to the Message Store: inlog and outlog. However, they are labelled ‘‘for further study’’.
- ISO allows a more flexible routing scheme (i.e., PRMDs can connect directly to more than one ADMD and are allowed to span national boundaries).
- ISO does not make interworking with X.400 (1984) mandatory whereas CCITT does, and makes the 1988 ‘‘stack’’ optional.

There are other smaller differences. For more details, see Manros (listed in **Referenced Documents**). However, the term ‘‘X.400’’ used in this document can mean either X.400 or MOTIS.

An implementation of the X.400 APIs (see XMHS listed in **Referenced Documents**) may provide support for the 1984 version of X.400, the 1988 version of X.400, or both.

1.2 DIRECTORY STANDARDS OVERVIEW

In recognition of an urgent need, the CCITT undertook a study in close cooperation with ISO/IEC to define an OSI Directory Service and associated protocols in the CCITT study period 1985-1988. The resulting CCITT X.500-series of Recommendations (X.500 - see **Referenced Documents** - corresponding to the ISO 9594 standard) were approved at the end of 1988.

The primary use of the Directory is in two areas: OSI communications (application name to address lookup) and X.400. However, the Directory is general-purpose and can be used for many other communications-related activities, such as simple telephone number lookup and postal address lookup.

X.500 defines a Directory with a tree-like structure of objects that relate to communications. The objects have attributes that describe them. Examples of object types are people, organisations, application processes and devices. Examples of attribute types are telephone numbers, street addresses, titles, serial numbers. The attributes are also encoded in a particular way. This is referred to as the Attribute Syntax. Examples of Attribute Syntaxes are Printable String, Octet String, Integer and Boolean.

The Directory holds information which it provides at the request of the user with the correct privileges. The *Directory Access Protocol* (DAP) allows various ways of accessing the information in the Directory. It is possible to use the Directory much like the White and Yellow Pages.

The Directory can also be distributed over many computer systems and, potentially, over wide geographical areas. The X.500 Recommendations define a set of ‘‘distributed procedures’’ for use among the pieces of the Directory. This is referred to as the *Directory System Protocol* (DSP). Fairly rigid rules are required to allow for the Directory to be distributed over wide areas and over different administrations.

Some functional elements were also defined. The *Directory User Agent* (DUA) is the software that acts on behalf of the user. The *Directory System Agent* (DSA) manages the *Directory Information Base* (DIB). As shown in **Figure 1-3**, the DAP is used between the DUA and the DSA, whereas the DSP is used between the DSAs.

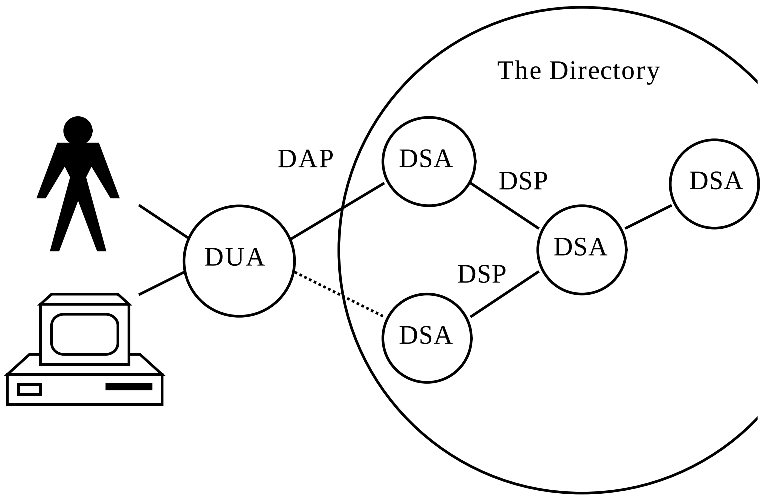
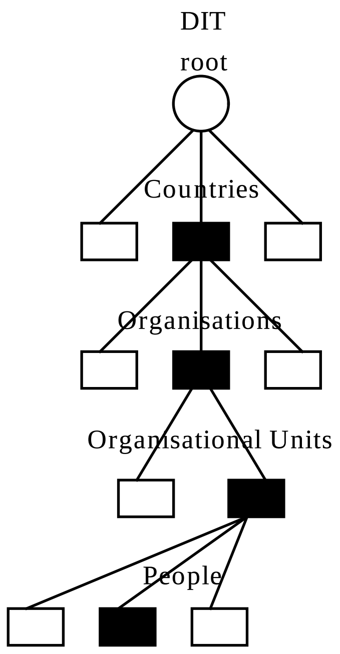


Figure **1-3.** DAP and DSP

Information in the Directory is organised as entries which are related to one another via a hierarchical structure known as the *Directory Information Tree* (DIT). Each entry is uniquely identified by its *Distinguished Name* (DN) which has a bearing on the location of the entry within the DIT (see **Figure 1-4**). Associated with each entry is an attribute termed the *Relative Distinguished Name* (RDN). The DN of a particular entry can be derived by concatenating the RDNs of each entry encountered when traversing the tree starting at the root until that particular entry is reached.



DIT	RDN	DN
root		{ }
Countries	C=GB	{C=GB}
Organisations	O=Telecom	{C=GB, O=Telecom}
Organisational Units	(OU=Sales, L=Ipswich)	{C=GB, O=Telecom, (OU=Sales, L=Ipswich)}
People	CN=Ian_Smith	{C=GB, O=Telecom, (OU=Sales, L=Ipswich), CN=Ian_Smith}

Figure **1-4.** Directory Information Tree

When a user (serviced by a DUA) needs some information, the DUA establishes communication with a DSA. The user can now request information from the Directory to which the DSA may or may not respond depending on the access privileges of the user. The DUA may be given, instead of the requested information, a referral to another DSA that is ‘‘suspected’’ to hold the information. Note that a DUA may choose, for expediency, to maintain communication with several DSAs simultaneously and DSAs may also maintain simultaneous communication with several other DSAs.

The **X/Open Directory Services API Specification** (XDS - see **Referenced Documents**) jointly published by XAPIA and X/Open provides an API to the Directory. Although XDS reflects the abstract services described in X.500 (see **Referenced Documents**), it is not a requirement that a product based on XDS actually implements the protocols described in X.500.

Chapter 2

Architectural Model

The APIs are functional interfaces. The term *service* denotes software that implements the functions of the API. The term *client* denotes software that uses the API, i.e., invokes or makes calls to the API (refer to **Figure 2-1**).

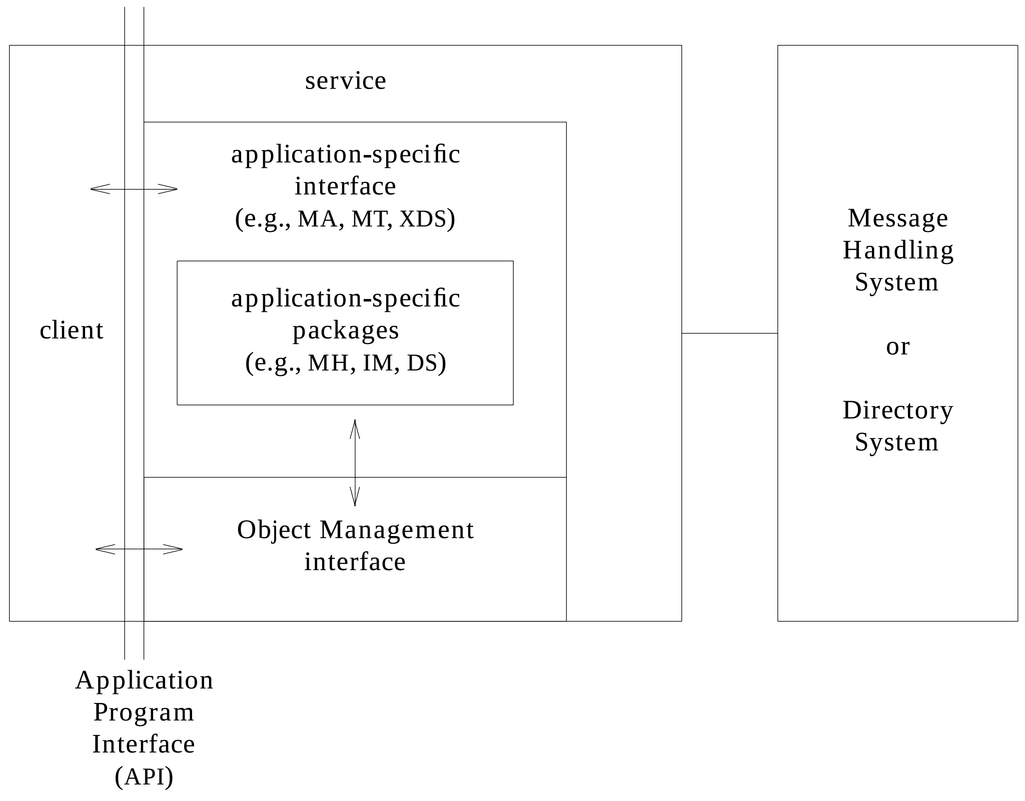


Figure **2-1.** General Conceptual Model

The service supplies a particular application-specific interface, such as:

- the X.400 *Message Access* (MA) interface (XMHS - see **Referenced Documents**)
- the X.400 *Message Transfer* (MT) interface (XMHS)
- the *X/Open Directory Services* (XDS) interface (XDS - see **Referenced Documents**)

An application-specific interface also includes one or more application-specific *packages* that define relevant classes for objects. (Refer to **Section 5.2.4**, **Packages** for the definition of a ‘‘package’’.) The service also supplies OSI *Object Management* (OM) interface functions (XOM - see **Referenced Documents**) that can be used to create, delete, access and update objects which are instances of classes defined in these packages. Examples of packages are:

- the Message Handling (MH) package (defined in XMHS)

- the Interpersonal Messaging (IM) package (defined in XMHS)
- the Directory Services (DS) package (defined in XDS)

A client may request certain *features* of the interface it would like to use for the duration of an API session. Features are requested in terms of packages and functional units that are defined in the respective API specifications. A *functional unit* (FU) is a collection of related interface functions.

Hence, the client can use the application-specific interface functions in conjunction with the OM interface functions to access services of a Message Handling or Directory system. For instance, a client can access an X.400 MHS using an MA or MT API, or can access an X.500 system using an XDS API, or perhaps, simultaneously access both an X.400 system and an X.500 system.

The ensuing Chapters elaborate on the MA interface, the MT interface, the XDS interface and the OM interface.

Chapter 3

X.400 APIs

The document XMHS (see **Referenced Documents**) defines two APIs that are of architectural importance in the construction of Message Handling Systems (MHSs). The X.400 APIs use the X.400 model for message handling. However, they do not require that X.400 is implemented by the service.

The X.400 Application API makes:

- the functionality of a message transfer system (MTS) accessible to a client acting as a message store or a user agent, and
- optionally, the functionality of a simple, message queuing facility called a ‘‘message store’’ accessible to a UA. It should be noted that such a ‘‘message store’’ is non-standard (i.e., different from the Message Store as defined by X.413 - see **Referenced Documents**).

The X.400 Application API defines the *Message Access* (MA) interface, which is used to convey information objects between the MTS and one of its users and thereby among its various users.

The X.400 Gateway API defines the *Message Transfer* (MT) interface which makes the X.400 MHS accessible to a client. The MT interface forms part of an MTA.

In the case where the interface can ‘‘understand’’ the structure of P2 contents, Interpersonal Messaging (IM) is said to be supported. The interfaces address the operational requirements of these activities, but they do not address system management requirements, e.g., those for security.

The X.400 Application API would typically be used where a client is some form of a User Agent that typically has a single address. The service is similar to the Message Transfer System Abstract Service described in MHS-1988:X.411 (see **Referenced Documents**). The X.400 Gateway API would typically be used to implement a ‘‘gateway’’; that is, a functional object which allows messages from one messaging system to be converted into the format and adhere to the conventions used by another messaging system. The client of the X.400 Gateway API typically has more than one address.

Another major difference is that the X.400 Application API provides full support for the functions in an MTS, whereas the X.400 Gateway API provides access to the raw transfer functions and the client is responsible for generating reports and trace information; in other words, it has to implement some MTA functions and has full control over these functions.

3.1 X.400 APPLICATION API/MESSAGE ACCESS INTERFACE

The X.400 Application API, or *Message Access* interface, gives access to the X.400 Message Transfer System (MTS).

The MA interface provides three queues which the service maintains even when it is out of contact with the client.

- Submission queue

  Using the API of the service, the client initiates placement of MTS-bound messages and probes in the Submission queue for subsequent handling by the MTS (see **Figure 3-1** and **Figure 3-2**). From the client's point of view, the maximum queue length is one.

- Delivery queue

  The service places inbound messages and reports in the Delivery queue for subsequent delivery to the client (see **Figure 3-1**).

- Retrieval queue

  Alternatively, the service places inbound messages and reports in the Retrieval queue for subsequent retrieval by the client (see **Figure 3-2**). Messages and reports in the Retrieval queue, unlike those in the Delivery queue, can be accessed at random. Once placed into the Retrieval queue, the messages and reports are considered by the MTS to have been ‘‘delivered’’.

The client can choose to use either the *Delivery Model* or the *Retrieval Model*. The Delivery Model would be used when the client is always available to take delivery of messages and reports. The Retrieval Model would be used when the client is not available at all times. A product that implements the Retrieval Model can take delivery of a message from the MTS without involvement from the client.

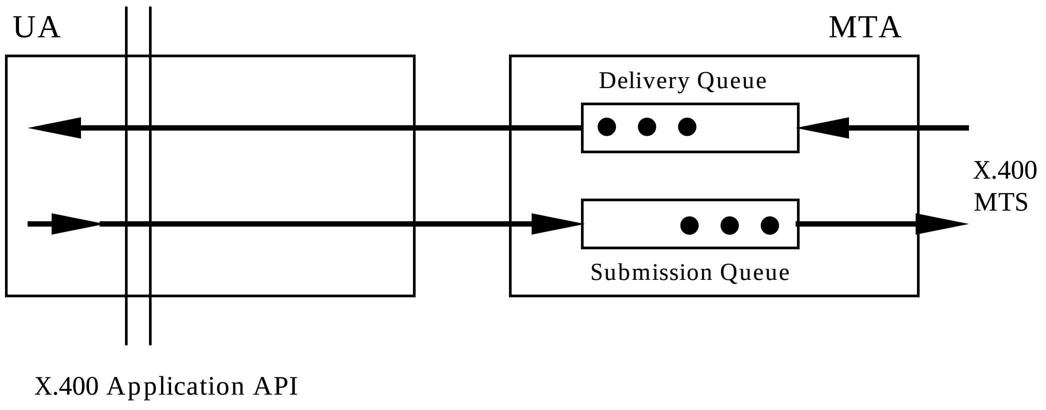


Figure **3-1.** X.400 Application API - Conceptual Model for Delivery

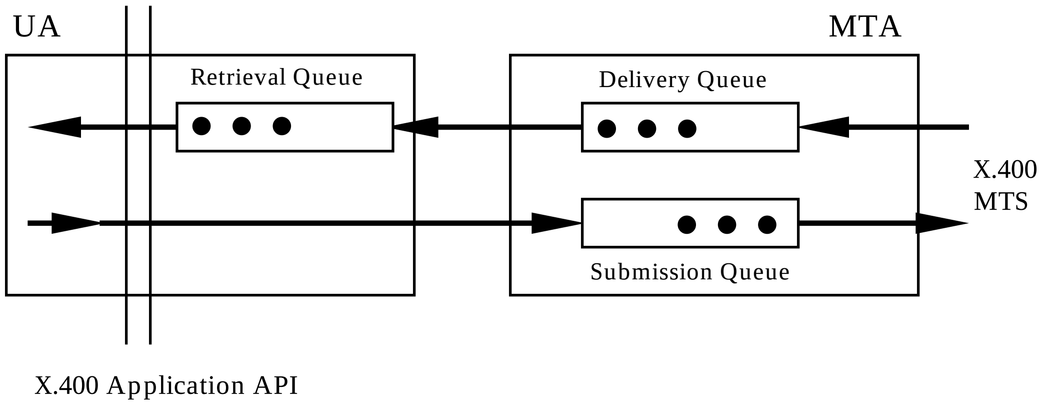


Figure **3-2.** X.400 Application API - Conceptual Model for Retrieval

The MA interface provides a general-purpose facility for implementing User Agents. A few specific UAs are defined in the standards:

- the *Interpersonal Messaging UA* (IM-UA),
- the *Electronic Data Interchange UA* (EDI-UA - see **Referenced Documents**).

The message content is indicated by the content type which is passed to the Message Transfer System upon submission of a message. The MTS uses this identifier to determine whether or not a message can be delivered at the destination(s). The content type generated by an IM-UA is commonly referred to as ‘‘P2’’. An MA interface product can (optionally) be packaged to provide special support for P2; i.e., it can assist the client in formatting P2 messages.

Other classes of UAs can be defined for special purposes: e.g., Library standards (LIB - see **Referenced Documents**) define a class of UAs for Inter-Library Loans, and the banking community has defined a class of UAs for transferring files (SWIFT - see **Referenced Documents**). However, the interface specification does not currently provide special support for these contexts, so the UA must format this kind of message completely before submitting it to the API.

3.1.1 MA Interface Functions

Functions	Description
MA-Open	establish a session with the MA interface
MA-Close	terminate a session with the MA interface
MA-Submit	submit a user message or probe
MA-Cancel-Submission	cancel delivery of message submitted with the deferred delivery option
MA-Size	determine the number of messages and reports in the delivery or retrieval queue for this user
MA-Wait	return when a message or report is available in the delivery or retrieval queue or when the specified interval lapses, whichever comes first
MA-Start-Retrieval	begin the retrieval of a user message or report
MA-Finish-Retrieval	conclude retrieval of a message object and use to delete items from the retrieval queue
MA-Start-Delivery	begin the delivery of a message or report
MA-Finish-Delivery	conclude delivery including (optional) delivery confirmations and non-delivery indications
OM interface functions	the functions that a client application can call to manipulate instances of OM objects (see **OSI-Abstract-Data Manipulation API (XOM) CAE Specification**)

3.1.2 MA Interface Functional Units

When initialising the MA interface, the client may specify either the Delivery or the Retrieval capability (but not both) be used for that session. The (optional) Retrieval capability is realised by the set of the above MA interface functions excluding the functions MA-Start-Delivery and MA-Finish-Delivery.

The MA interface may implement various functional units, of which Basic Access is mandatory:

MA Interface Functional Units	MA Interface Functions
Basic Access	MA-Open MA-Close OM interface functions
Submission	MA-Submit MA-Cancel-Submission
Delivery	MA-Size MA-Wait MA-Start-Delivery MA-Finish-Delivery
Retrieval	MA-Size MA-Wait MA-Start-Retrieval MA-Finish-Retrieval

3.1.3 Possible Product Configurations

In general, an MA interface implementation includes an MTA and, optionally, some utilities for generating one type of content, e.g., P2 content for Interpersonal Messaging.

Implementations of the MA service may also optionally:

- implement P2
- implement P3
- implement P7
- implement the Lower Layers (OSI) protocols

In addition to the actual MA interface functions, functions from the OM interface (refer to **Chapter 5**, **OSI Object Management API**) must also be available. In practice, this would normally be supplied by the same vendor that supplied the MA interface.

The possible configurations are shown in **Figure 3-3** in abstract form - no assumptions are made about the configuration of real software modules. The actual configuration is determined by the supplier of the MA interface product.

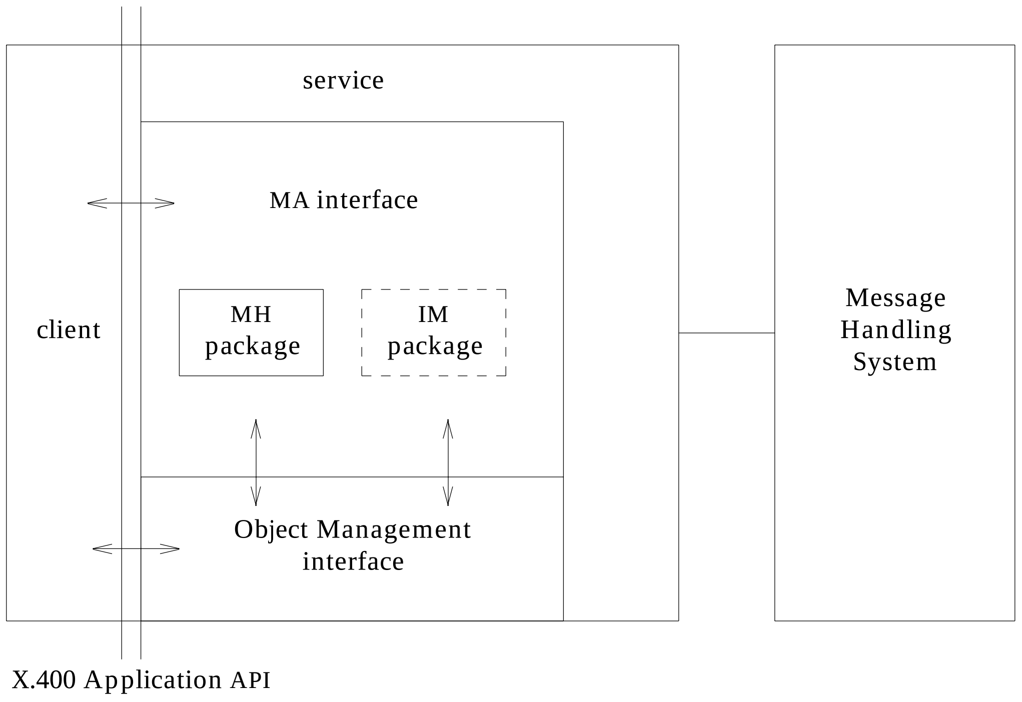


Figure **3-3.** X.400 Application API - Possible Product Configurations

The MA interface product specification should include how X.400 service elements are supported and how the MA interface is supported. See also Conformance details in XMHS (see **Referenced Documents**).

3.2 X.400 GATEWAY API/MESSAGE TRANSFER INTERFACE

The X.400 Gateway API, or Message Transfer interface, allows the construction of a gateway, i.e., a process or program able to perform :

- the appropriate protocol conversion (message format translation), and
- mail-service mapping

between an X.400 and a non-X.400 electronic mail system.

The interface that implements this functionality is called *Message Transfer* (MT). A client that uses the MT interface may realise a mail system gateway.

The MT interface is designed to simplify the implementation of gateways between two different messaging handling systems where one is likely but not necessarily X.400-based.

The MT interface deals with two queues which the service maintains even when it is out of contact with the client (see **Figure 3-4**).

- output queue
  into which the client places outbound messages, probes and reports for subsequent action by the service.
- input queue
  into which the service places inbound messages, probes and reports for subsequent action by the client.

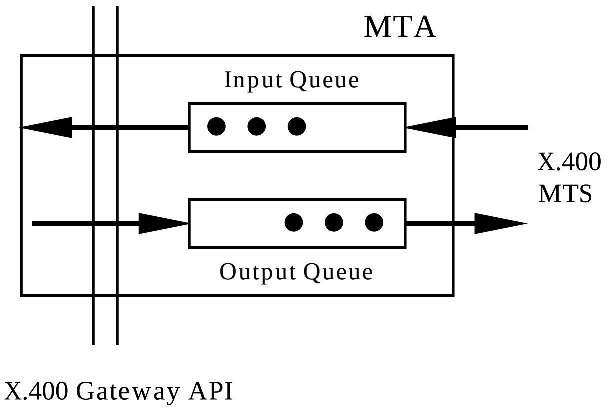


Figure **3-4.** X.400 Gateway API - Conceptual Model

The functions of the MT interface are implemented by the service. Jointly, the client and the service form an MTA which can generate or receive messages, probes and reports at the service level. The output is X.400-compliant (known as P1 in X.400).

A client can, for instance, be a part of another electronic mail system and serve as a gateway between the two systems. The MT interface gives the client full control over the generation and handling of all messaging objects (i.e., messages, reports and probes). A product that implements the MT interface may optionally be packaged with support for Interpersonal Messaging (IM) allowing clients to work with the details of IM message types through the OM interface.

3.2.1 MT Interface Functions

Function	Description
MT-Open	establish a session with the MT interface
MT-Close	terminate a session with the MT interface
MT-Transfer-Out	add a messaging object (message, probe or report) to the output queue
MT-Start-Transfer-In	begin the transfer of a (unreserved) messaging object in the input queue
MT-Finish-Transfer-In	conclude one or all ongoing transfers and if requested by the client, remove the messaging objects from the input queue
MT-Size	determine the number of messaging objects (messages, probes and reports) in the input queue
MT-Wait	return when an object is available in the input queue or when the specified interval lapses, whichever comes first
OM interface functions	the functions that a client application can call to manipulate instances of OM objects (see **OSI-Abstract-Data Manipulation API (XOM) CAE Specification**)

3.2.2 MT Interface Functional Units

The MT interface may implement various functional units, of which the Basic Transfer Functional Unit is mandatory:

MT Interface Functional Units	MT Interface Functions
Basic Transfer	MT-Open MT-Close OM interface functions
Transfer Out	MT-Transfer-Out
Transfer In	MT-Size MT-Wait MT-Start-Transfer-In MT-Finish-Transfer-In

3.2.3 Possible Product Configurations

The MT interface provides access to a set of X.400 services normally implemented as an MTA and may optionally be packaged with some tools for generating Interpersonal Messages.

The possible configurations are depicted in **Figure 3-5**. No assumptions are made about the configuration of real software modules. The actual configuration is determined by the supplier of the API product.

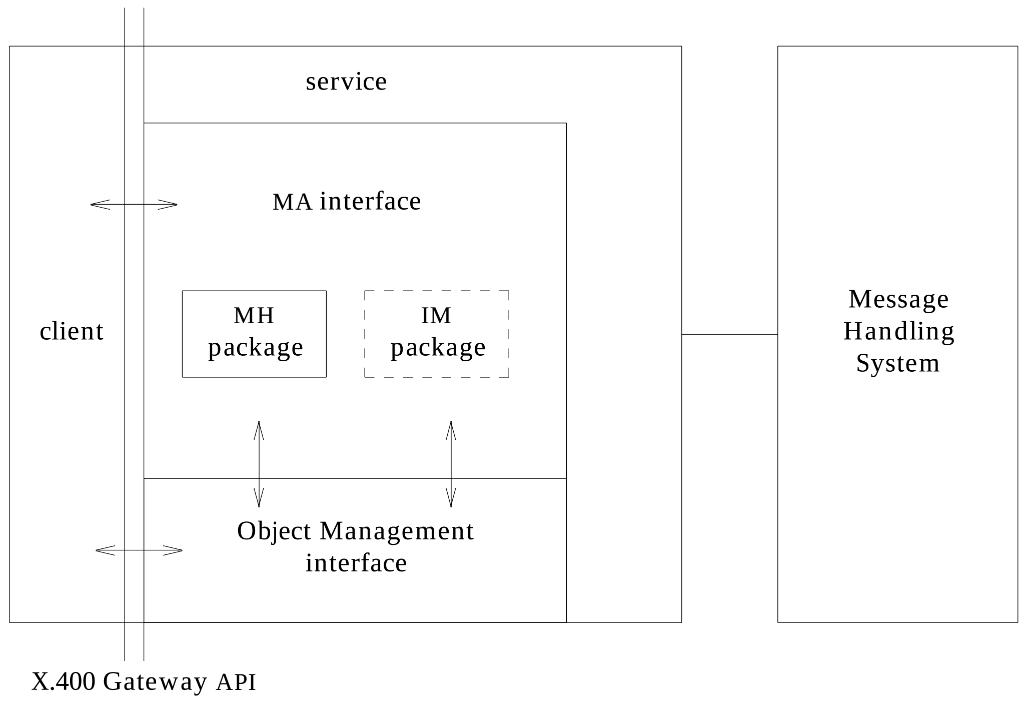


Figure **3-5.** MT Interface - Possible Product Configurations

In addition to the actual MT interface, the OM interface functions must also be available. In practice, this would normally be supplied by the same vendor that supplies the MT interface.

The API may also be extended by non-X.400 functions that are outside the scope of this document.

The MT interface product specifications should include how X.400 service elements are supported and which of the MT interface Functional Units are supported. The IM package may optionally be included.

Chapter 4

Directory API

The X/Open Directory Services (XDS) API is designed as an operational interface to the X.500 Directory. The XDS interface is the interface between the *Directory User Agent* (DUA) and an application program. Through this XDS interface, the client application program can access the Directory in order to make queries and updates.

The XDS interface is designed so that each Abstract Service defined by the Directory Service standards maps to a single function call (such as Read, Abandon and List). Due to the nature of some of these operations, the XDS interface provides support for both synchronous and asynchronous function calls.

When reading XDS (see **Referenced Documents**), note that the term ''attribute'' is used in two contexts. The unqualified term ''attribute'' is used to describe an object (or ''entry'') in the Directory (X.500 - see **Referenced Documents**). In contrast, the term ''OM attributes'', used in the Object Management context, is described in **Section 5.2**, **Terms and Concepts**. In addition, the terms ''class'' and ''object'' denote the constructs defined in X.500, while the terms ''OM class'' and ''OM object'' denote the OM constructs.

However, this document (especially **Chapter 5**, **OSI Object Management API**) uses the unqualified terms ''object'', ''class'' and ''attribute'' to refer to the OM constructs.

4.1 XDS INTERFACE FUNCTIONS

Function	Description
DS-Initialize	initialise the XDS interface
DS-Version	negotiate features of the interface and service
DS-Shutdown	shutdown the XDS interface
DS-Bind	open a session with the Directory
DS-Unbind	close a directory session
DS-Receive-Result	retrieve the result of an asynchronously executed operation
DS-Abandon	attempt to abandon the result of a pending, asynchronously executing operation
DS-Add-Entry	add a leaf entry to the DIT
DS-Compare	compare a purported attribute value with the attribute value stored in the Directory for a particular entry
DS-List	enumerate the immediate subordinates of a particular Directory entry
DS-Modify-Entry	perform an atomic modification on a Directory entry
DS-Modify-RDN	modify the relative distinguished name (RDN) of a leaf entry
DS-Read	query information on an entry by name
DS-Search	find entries of interest in a portion of the Directory Information Tree
DS-Remove-Entry	remove a leaf entry from the Directory Information Tree

4.2 POSSIBLE PRODUCT CONFIGURATIONS

X.500 products can be configured in a variety of ways. The XDS must always provide the functions of a DUA, but an actual DSA may or may not be provided. Note that XDS states that DAP and DSP are not mandatory. This could mean that an XDS product could be provided without actual support for distributed directories in the X.500 sense.

Vendors of XDS products should specify which Object Classes, Attribute Classes and Attribute Syntaxes the product supports and how X.500 is supported. This includes:

- support for DAP
- if a DSA is provided, whether it can work as a first-level DSA and whether DSP is supported
- ability to maintain multiple concurrent DUA-DSA connections
- amenability to defining new Directory attribute types and classes

Chapter 5

OSI Object Management API

Any of the above application-specific APIs must be used in conjunction with the general purpose OSI Object Management (OM) API. The OM interface is designed to be used with application-specific APIs that provide OSI services to allow the transfer of Abstract Syntax Notation One (ASN.1) protocol elements in an application-independent fashion.

The objects addressed by the information architecture are those based on ASN.1 (see Steedman, X.208 and X.209 listed in **Referenced Documents**). By providing tools for manipulating ASN.1 objects, the OM interface shields the client from much of the complexity of encoding and decoding ASN.1 elements using the ASN.1 Basic Encoding Rules (BER). While the OM interface allows manipulations of ASN.1-based objects in general, an OM interface product is not required to support BER encoding and decoding of objects in general. The BER encoding supported by an OM interface product is limited to the extent required for the encoding and decoding of those objects specified in the *packages* supported (see **Section 5.2.4**, **Packages** for definition of package).

The Object Management API provides a platform on which more than one application-specific API can be built. An application program must format its data into objects and then submit (or retrieve) these objects using programmatic ‘‘calls’’ which are standardised by the Object Management interface.

The Object Management API consists of:

- the syntactical definition of an *OM object*,
- the functions that a client application can call to manipulate instances of OM objects.

The OM API presents to the programmer a uniform model for information based on the concept of *classes*, i.e., groups of similar information objects. A *class* is described by a collection of OM attributes; each attribute consists of one or more values, each of which is characterised by a type and an OM syntax (e.g., Integer, Boolean and String). Objects are used to convey information between the client and the service.

The representation-hiding that the OM interface provides alone is inadequate to fully meet the needs of environments supporting several application-specific APIs, e.g., where an X.400 Application API and an XDS API can coexist in the same run-time environment. The different APIs may impose different, and even conflicting, requirements on the internal representations of objects and they might even be implemented by different vendors.

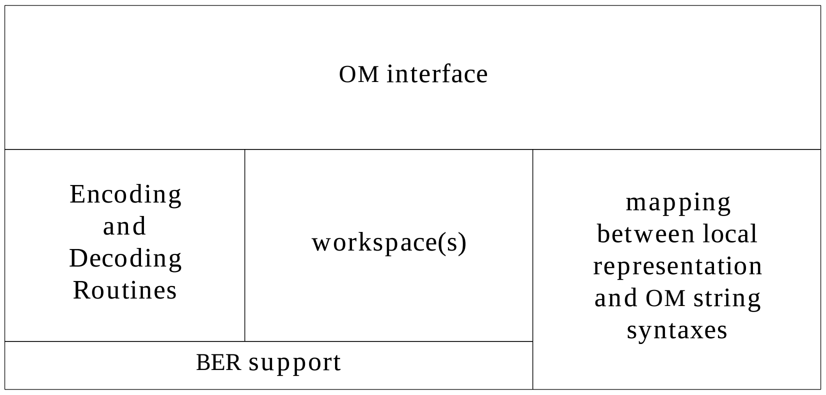
However, the OM interface does allow any number of OM interface implementations to coexist, each representing objects differently. This is accomplished by means of *workspaces*. A *workspace* is the means by which a client application can share data among different APIs.

Since all the application-specific APIs have the Object Management API in common, a client implementor need only generate programs meeting that portion of the interface once, with the additional software which creates the specific (X.400 Gateway, X.400 Application or XDS) API(s) as required. In certain implementations, this commonality of software usage may result in lower usage of memory space at the workstation level.

The ensuing subsections cover possible OM API product configurations, some important terms and concepts used in the OM API, the OM interface functions and use of the OM interface.

5.1 POSSIBLE PRODUCT CONFIGURATIONS

An implementation of the OM interface must support the ASN.1 Basic Encoding Rules (BER) and one or more workspaces.


OM interface
Encoding and Decoding Routines
workspace(s)
mapping between local representation and OM string syntaxes
BER support

The specifications for an OM API implementation must define:

- the mappings between its local character set representation and the various OM string syntaxes
- any intermediate data types used
- the maximum length of a string returned by the *OM-Get*( ) function (and this must be at least 1024)
- whether or not exceptions will be reported if an object supplied as an argument is not consistent with the definition of its class

5.2 TERMS AND CONCEPTS

The following subsections briefly introduce some important terms and concepts used in the OM API.

5.2.1 Objects

This subsection describes *OM objects*. (The term ‘‘object’’ is used here in the context of the OM API.) The principal purpose of the OM service is to create, examine, modify and destroy complex information objects under the client’s direction. The client and the service can exchange objects in whole or in part.

Public/Private Objects

Objects are of two kinds: private and public.

Private A *private object* is represented in a fashion that is opaque to the client and thus unspecified. Thus, the client accesses private objects only indirectly, i.e., by means of the OM interface functions.

Public A *public object* is represented by a data structure whose format is well defined: a list of descriptor data structures, each describing an attribute value. This data structure is understood by the client and the service, and thus is the main means by which the client and the service exchange the attribute values.

A public object is created, examined, modified and destroyed directly, i.e., by means of programming language constructs (however, restrictions might apply for operations on service-generated public objects - see below).

The use of public objects simplifies application programs, allowing the program to statically define and instantiate objects instead of constructing objects dynamically using the OM interface function calls.

Public objects themselves may be either:

— client-generated, or

— service-generated.

Client-generated public objects are created by the client in storage provided by the client who is responsible for managing the storage.

Service-generated public objects are created by the service in storage provided by the service. The client destroys a service-generated public object and releases its storage by applying the *OM-Delete*( ) function to it. A service-generated public object is unaffected by the destruction of the workspace that generated it.

A summarised comparison of the properties of private and public objects is given in the table below.

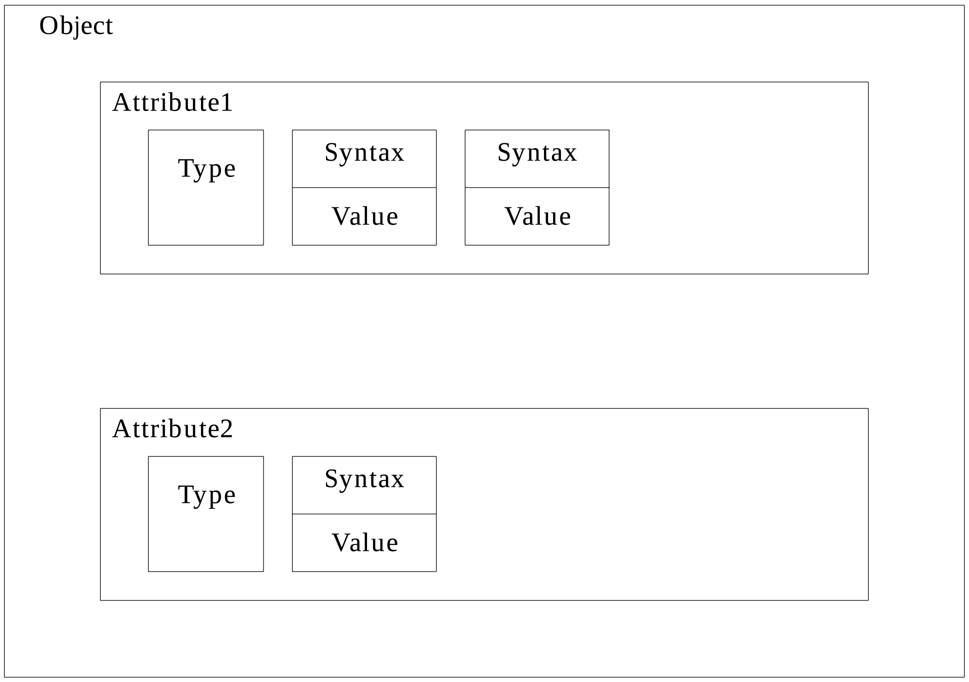
Private	**Public**
- representation is specific to the implementation	- representation is defined in the API specification
- not directly accessible by client	- directly accessible by client
- manipulated by client using OM functions	- manipulated by client using programming constructs
- created in storage provided by the service	- said to be service-generated if created by the service in storage provided by the service, or - said to be client-generated if created by the client in storage provided by the client
- cannot be modified by the client directly (except via the service interface)	- if client-generated, can be modified directly by the client, whereas if service-generated, cannot be modified directly by the client (except via the service interface)
- storage allocated and released by the service — created in workspace — destroyed using *OM-Delete*( ) or when the workspace associated with the object is destroyed	- storage allocated and released by: — the service if object is service-generated — the client if object is client-generated

5.2.2 Object Attributes

An object comprises zero or more information items called attributes.

Note: These ‘‘attributes’’ pertaining to the objects defined in the Object Management specifications are not the same as the ‘‘attributes’’ as used in the Directory.

An *attribute*, in turn, comprises an integer denoting the *type* of the attribute and one or more information items called *values*, each accompanied by an integer denoting the *syntax* of the value.

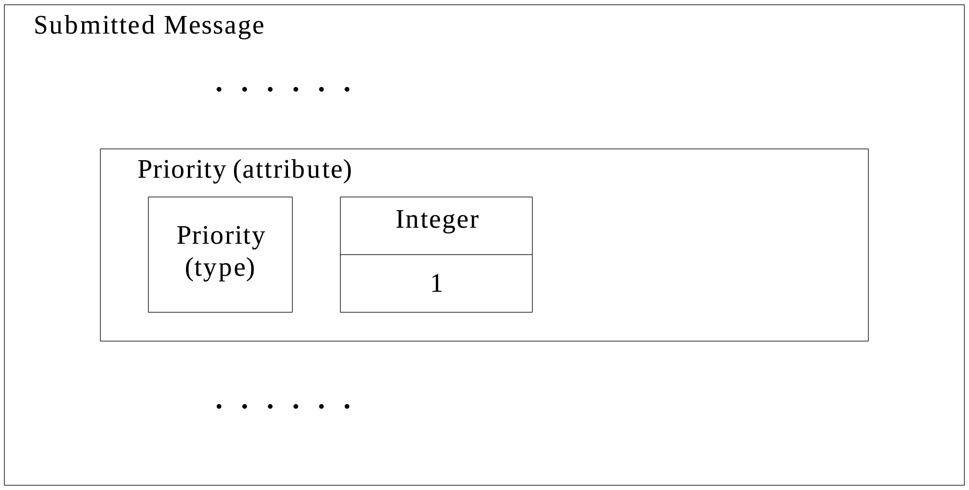


A *type* is a category into which all the values of an attribute are placed on the basis of its purpose. Some attributes may either have a single value or multiple values. The attribute type is used as the name of the attribute.

A *syntax* is a category into which a value is placed on the basis of its form.

A *value* is an information item which can be viewed as a characteristic or property of the object of which it is a part.

For example (see the following figure), in a *Submitted-Message* object (from the Message Handling package XMHS - see **Referenced Documents**), one attribute type may be ‘‘priority’’ which has a syntax of ‘‘Integer’’ and a (single) value of ‘‘1’’. Note that the attribute type is used as the name of the attribute.



The value of an attribute of an object may, in turn, be an object. In this case, the latter object is said to be a *subobject* of the former; and the former object is said to be a *superobject* of the latter. For instance, the *Originator-Name* attribute of a *Submitted-Message* is itself a complex structure of the *OR-Name* class.

The client and service exchange values by means of descriptors. A *descriptor* normally comprises a value and the integers that denote the value's syntax and type.

Syntaxes and types are denoted by integers although the scope of these integers differ.

- The scope of the integers for syntaxes is global, and these integers are defined in the OM API document (see **referenced documents**).
- The scope of the integers for classes is a package (see **Section 5.2.4**, **Packages** for the definition of a package). Types are defined and assigned integers by OM applications.

From the point of view of the client, the attributes of an object are unordered (although they may actually be position-dependent in the encoded object). However, the values of the attributes are ordered; the position of the first value is zero and positions of successive values are successive positive integers.

5.2.3 Classes

Objects are categorised into classes based on their purpose and internal structure.

An object (e.g., a message) is said to be an *instance* of its class (e.g., Message). A class is characterised by attribute types that may appear in its instances. A *class* is denoted by an ASN.1 object identifier and this object identifier, called the Class attribute, is an attribute of every instance of the class.

Class Hierarchy and Inheritance Properties

The classes are related to each other, forming a tree hierarchy whose root is a special class, Object, and each of the other classes is the immediate *subclass* of precisely one other class (i.e., its immediate *superclass*). This tree structure, known as the *class hierarchy*, is important because of the inheritance property. The class hierarchy is defined in the API documents and cannot be modified by a program.

The attribute types that may exist in an instance of a class but *not* in an instance of its immediate superclass are said to be ‘‘specific’’ to that class.

In fact, the attribute types that may appear in an object are those specific to its class as well as those specific to each of its superclasses (through inheritance).

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

Abstract Versus Concrete Classes

Classes may be abstract or concrete.

An *abstract* class is one of which instances are disallowed. 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.

In contrast to abstract classes, instances of *concrete* classes are permitted. However, the definition of each concrete class may also indicate that the client not be allowed to create instances (e.g., the *Session* class in the XDS interface); in this case, instances can only be created as a result of an application-specific function. It would then be an error for a client to attempt to create an object of such a (function-declined, concrete) class or of an abstract class.

Consider the following alternative means of defining the classes, *List-Info* and *Read-Result*, used in the Directory.

Case 1: (Using an Abstract Class)

As defined in XDS (see **Referenced Documents**), the concrete classes *List-Info* and *Read-Result* are subclasses of the abstract class, *Common-Results*.

The OM attributes of the abstract class *Common-Results* are:

- Alias-Dereferenced
- Performer

The OM attributes of the class *List-Info* are those it inherits from its superclass *Common-Results* as well as those specific to this class, namely:

- Object-Name
- Partial-Outcome-Qualifier
- Subordinates

The OM attributes of the class *Read-Result* are those it inherits from its superclass *Common-Results* as well as those specific to this class, namely:

- Entry

Case 2: (Without Using an Abstract Class)

Alternatively, the *List-Info* and *Read-Result* classes may have been defined without defining its superclass, abstract class, *Common-Results*. In this case, the OM attributes for the class *List-Info* would be:

- Object-Name
- Partial-Outcome-Qualifier
- Subordinates
- Alias-Dereferenced
- Performer

whereas the OM attributes for the class *Read-Result* would be:

- Entry
- Alias-Dereferenced
- Performer

Specification of a Class

A class is defined in terms of these elements:

- class name (denoted by an object identifier)
- identity of its immediate superclass (important for determining inherited attribute types)
- definitions of the attribute types specific to the class
- constraints on the attributes (see the next paragraph)
- whether the class is abstract or concrete

In the class definition sections of XDS (and XMS), a class is concrete unless otherwise stated. For XMHS and XOM, information about the identity of the superclasses of a class as well as whether the class is abstract or concrete must be ascertained from the ‘‘Class Hierarchy’’ charts.

Constraints on Attributes in a Class

The specification of a class may impose arbitrary constraints on its attributes. For instance, the most common of these include the constraints to:

- restrict the syntaxes permitted for values of an attribute (often to a single syntax)
- restrict the particular values to a subset of those permitted by the syntax
- require one or more values of the attribute (i.e., a *mandatory* attribute)
- allow either zero or more values of the attribute (i.e., an *optional* attribute)
- permit multiple values, perhaps up to some limit known as the *value number constraint*
- restrict the length of strings (in octets), up to a limit known as the *value length constraint*

Constraints may affect multiple attributes at a time; e.g., a rule that only one of a set of several attributes may be present in any OM object.

5.2.4 Packages

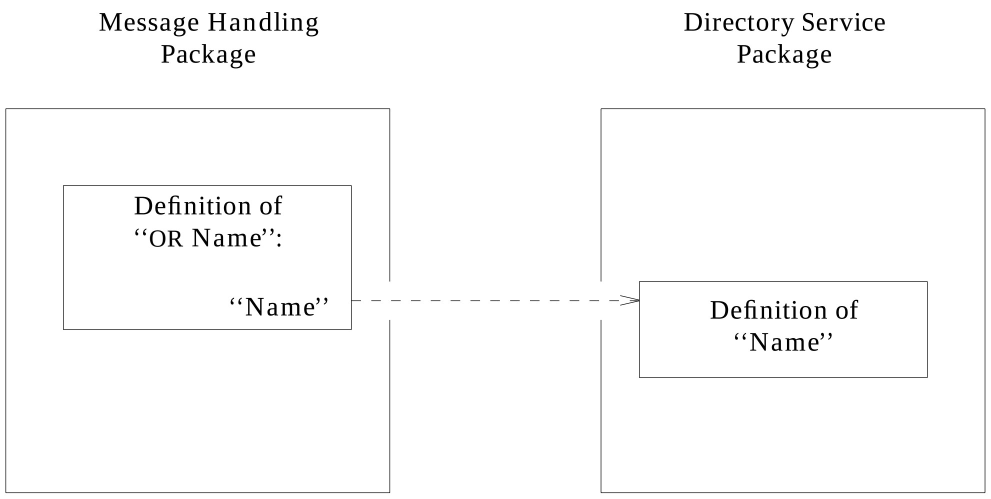
A *package* is a collection of classes that are grouped together by an API specification because these classes are functionally related.

A package is uniquely identified by an OSI *object identifier* that is assigned by the specification.

A package defines the scope of the integers denoting the attribute types specific to its classes. Thus, within a package, these integers shall be distinct.

Closure of a Package

A class may be defined to have an attribute whose class is defined in some other package. This is done to share definitions and to avoid duplication. For instance, the Message Handling package defines a class called *OR Name* (refer to XMHS, see **Referenced Documents**). This class has an attribute whose syntax is an object of class, *Name*, which is defined in the Directory Service package (XDS).



In addition, a class may be a subclass of a class in another package.

These relationships lead to the concept of a Package-Closure.

A *Package-Closure* is the set of classes which need to be supported to ensure the ability to create all possible instances of all classes defined in the package.

Definition of a Package

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

- package name (an object identifier)
- definitions of one or more classes belonging to the package
- identification of zero or more concrete classes in the package to which the *OM-Create*( ) function applies (in every implementation of the service)
- identification of zero or more concrete classes in the package to which the *OM-Encode*( ) function applies (in every implementation of the service)
- (for the convenience of the reader) explicit identification of zero or more classes in other packages that appear in the closure of the package

5.2.5 Workspaces

Two application-specific APIs (e.g., MA interface, MT interface, XDS) may involve the same class, although the two APIs may employ different implementations of the OM service. For instance, when supplied by different vendors, the two implementations may represent private objects differently.

For the client to use both application-specific APIs, it 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 may be converted from one internal format to another (e.g., using OM encoding and decoding functions, or using *OM-Put*( ) and *OM-Get*( ) functions). Such interworking between service implementations is achieved by means of *workspaces*.

The OM interface includes functions for copying and moving objects from one workspace to another, provided that the objects' classes are associated with both workspaces.

Details of the representation of private objects and of the implementation of the functions that are used to manipulate them are not specified since they are not of concern to the client programmer. However, the programmer sometimes needs to be aware of which implementation is being used for a particular object.

Recall the example **Closure of a Package** in the preceding **Section 5.2.4**, **Packages**, where the class, *Name*, is used in both the Message Transfer Service and the Directory Service. If a client uses both services and the two services use different internal representations of objects (perhaps because the services are implemented by different vendors), then it is necessary for the client to specify which implementation should create a (private) *Name* object. This is done by means of a workspace.

The notion of a workspace also includes the storage used to represent objects and management of that storage. For more details on how workspaces are implemented, refer to XOM (see **Referenced Documents**).

The client must obtain a workspace that supports a class before it is able to create any objects of that class. The workspaces are returned by functions in the appropriate specific API service. For instance, the *MT-Open*() function returns a workspace that supports, in particular, the Message Handling package (XMHS - see **Referenced Documents**).

Some implementations may support additional packages in a workspace. For instance, vendors may provide classes for directory attribute types in the Basic Directory Contents package as well as the Strong Authentication package (XDS).

Another important case is where two or more services are supported by the same implementation. For instance, in an implementation that supports both the Directory Service and the Message Transfer Service, the workspaces returned by the functions *DS-Initialize*( ) and *MT-Open*() are likely to be the same one. The client need not be aware of this.

5.3 OM API - INTERFACE FUNCTIONS

The OM API defines general purpose functions that allow the manipulation of objects. These functions are summarised as follows:

Function	**Description**
OM-Create	create a private object that is an instance of the specified class
OM-Delete	delete a private or service-generated, public object
OM-Instance	test if the object is an instance of the specified class (including the case when the object is a subclass of that class)
OM-Get	create a new public object that is an exact but independent copy of an existing private object; certain exclusions and/or syntax conversion may be requested for the copy
OM-Put	place or replace, into the target private object, copies of the attribute values of the source private or public object
OM-Read	read a segment of a string attribute of a private object
OM-Write	write a segment of a string attribute of a private object
OM-Copy	create, into the specified workspace, an independent copy of an existing private object and all of its subobjects
OM-Copy-Value	replace an existing attribute value or insert a new value in the target private object with a copy of an existing attribute value found in the source private object
OM-Encode	create a new private object that encodes, based on the specified encoding rules, an existing private object for the purpose of moving between workspaces, transport over a network or storage in a file
OM-Decode	create a new private object that is an exact but independent copy of the object that an existing private object encodes
OM-Remove	remove and discard values of an attribute of a private object.

5.4 USING THE OM INTERFACE

OM objects are used to represent data collections used in the interface, such as parts of a message, or the results of a messaging or directory access function.

An instance of a *subclass* can be used wherever a particular OM class is needed. This means both that the client can supply a subclass and that the service can return a subclass. For example, the client can submit messages in any format which is defined as a subclass of the class *Submitted-Communique* and the service returns all results of the submit function in any subclass of the *Submission-Results* class (XMHS - see **Referenced Documents**).

Since the service may return a subclass of the specified OM class, the client should always use the *OM-Instance*( ) function when checking the OM class of an OM object, rather than testing the value of the Class OM attribute.

The *subclassing* mechanism can be used to allow different specialisations of a class to be used in the same manner in an interface. To further specialise interface classes, additional packages may be defined for specific application domains or for extensions in specific vendor products. Hence, when the client supplies a subclass of a specified OM class as an argument, the service either will recognise the subclass as an OM class of a service-supported package or will ignore all OM attribute types which are not permitted in that OM class. The client can generally supply either a public object or private object as an argument of the interface functions. There are exceptions, such as the *Session* argument of the MA, MT, XDS interfaces, which must be a private object. The interface will always return private objects. The client can convert these into public objects by calling *OM-Get*( ), if required.

Note that public objects returned by *OM-Get*( ) are read-only and must not be modified in any way.

When private or service-generated public objects are no longer required, *OM-Delete*( ) should be used to release their storage. Note, for instance, that *OM-Put*( ) makes copies of attributes of a source object for insertion or replacement in the target object. Hence, in the process of constructing an object using *OM-Put*( ), it is advisable to delete (temporary) objects when they are no longer required.

The object identifiers for classes needed by a client program have to be exported and imported, e.g., using these macros from **<xom.h>** (XOM - see **Referenced Documents**):

OM_EXPORT( )	which allocates memory for class constants required within a compilation unit
OM_IMPORT( )	which makes available the class constants within a compilation unit

Chapter 6

Programming Examples Using the APIs

6.1 INTRODUCTION

In this Chapter, we look at how the APIs can be used in practice. We also present a few programming examples to illustrate the use of the MA and MT interfaces. Sample programs that use the XDS interface are provided in XDS (see **Referenced Documents**).

X.400 MA interface	-	submitting a message
	-	receiving a message
X.400 MT interface	-	transferring a message out
	-	transferring a message in
XDS	-	reading a Directory entry synchronously (refer to XDS, see **Referenced Documents**)
	-	updating a Directory entry in the asynchronous mode (refer to XDS, see **Referenced Documents**)
OM interface	-	usage of the OM interface functions in all the above examples

Note that in the C programming examples in this chapter, compatibility with ISO C has not been verified.

6.2 MA INTERFACE FOR IM USER AGENTS

A user agent (UA), generally, uses some interactive interface or programmatic interface to accept input for composing messages to be submitted to the mail system or to manipulate and render messages. For composing a message, we assume that values for components of the X.400 message are somehow available.

An X.400 IM User Agent

The term ''UA'' is often misused. By ''UA'' in this context, we mean the set of software modules to compose and render X.400 messages. This section treats the X.400 User Agent in abstract terms. Actual product implementations may vary significantly in architecture.

The most common type of User Agent is the IM or ''P2'' UA. For this kind of UA, the client application would typically contain the software components shown in **Figure 6-1**. The Man-Machine Interface (MMI) could be a human interface or it could be an interface to another software module, for instance, a message-driven application.

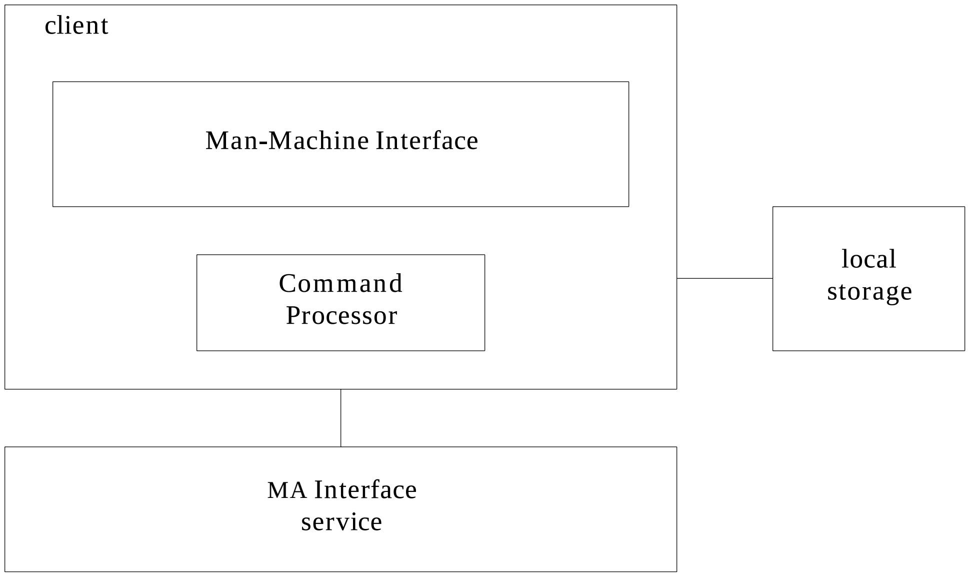


Figure **6-1.** An Example Client Application Using the MA Interface

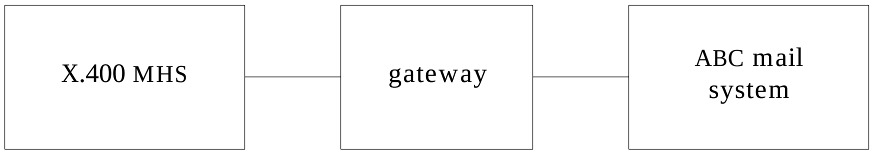
The Command Processor contains various routines to process commands issued by the user of the MMI. These could, for instance, be commands to read, edit, compose or send messages.

The above example assumes that the service provides the Interpersonal Messaging (IM) package (XMHS - see **Referenced Documents**); optionally, the client can encode parts of a message.

An IM-UA would also typically have some sort of local message storage, and finally, some sort of interface to the MTS is required. This could be a communications interface (perhaps using P3) or a local programmatic interface. In our case, the MA interface serves this purpose.

6.3 MT INTERFACE FOR GATEWAYS

The capability to send and receive X.400 messages can be extended to non-X.400 mail users by means of *gateways*. An implementor of a non-X.400 mail system can, of course, choose to completely replace the internals of such a product with native X.400 services, but in many cases, this is not the chosen method. The capability to send and receive X.400 messages can be extended to non-X.400 mail users by inserting a gateway between the non-X.400 world and the X.400 world. A gateway is essentially an Access Unit (AU) in X.400 terminology.



From the point of view of X.400, the users of a non-X.400 e-mail system are treated as belonging to a single MTA, with the non-X.400 mail server assuming full responsibility for routing messages to its individual users, once received from the X.400 MTA.

The gateway must understand the two mail systems completely in terms of:

- the communications systems
- the transfer systems
- the message syntaxes
- translation from one message syntax to the other

A product based on the MT interface takes care of the X.400 side of the gateway. The gateway implementor must take care of the non-X.400 communication system, transfer systems and message syntax. In addition, the implementation has to address the translation of syntax and facilities between the non-X.400 mail system and those required by the API.

A full gateway must then have access to both mail systems. The MT interface provides the access to important X.400 facilities (and through these, the communications system used by that particular product) as well as a ‘‘toolkit’’ of utilities to help the designer produce the gateway. The work of the designer is then substantially reduced to taking care of the translation from one mail system to the other. This task can be divided into the following activities:

- Translation of syntax

  This is one of the stronger aspects of the API. It shields the designer from much of the complexity of the syntax used in X.400 (ASN.1 Basic Encoding Rules - BER). The API specifies how to present the data. The API OM package can be used to construct X.400 messages in the proper ASN.1 format.

- Conversion of addresses

  Non-X.400 mail systems may use addresses quite different from those of X.400. In this case, the gateway might need to convert one type of address to another, ideally in a transparent manner. This means that users of one mail system should be able to send mail to users on the other mail system in a similar way as sending to users within their own particular mail system. An Appendix in XMHS (see **Referenced Documents**) hints at how this can be done.

- Facilities mapping

  X.400 is very rich in facilities. Among other facilities, the standards define several types of reports, various delivery options and extensive cross referencing. One of the issues the gateway has to handle gracefully is how to convey as much of the original information as possible from one side of the gateway to the other. This particular function of the gateway is not trivial to design and can impact the overall use of the entire messaging environment. Consider, for instance, the handling of the X.400 ‘‘Report Request’’ in the gateway between an X.400 and a non-X.400 system that does not support any equivalent of this service. Since the non-X.400 mail system does not support Delivery Notification, the gateway will have to issue this mandatory X.400 service element back to the X.400 user. The X.400 user will now - incorrectly - assume that the message has been delivered to the recipient’s UA, although it may have to go through several non-X.400 transfer agents before reaching its destination, during which it *could* get lost.

6.4 SOME GENERAL GUIDELINES

This section highlights some general guidelines that are useful when writing client application programs that use the APIs.

In the API specification documents, packages are described in chapters organised under these headings:

- Class Hierarchy
- Class Definitions
- Syntax Definitions
- Declaration Summary

The ‘‘Class Hierarchy’’ depicts the hierarchical organisation of classes which are elaborated in the ‘‘Class Definitions’’ section, where they are listed in alphabetical order.

The ‘‘Class Definitions’’ indicate how to build objects that are to be used as arguments to the API functions.

The ‘‘Class Definitions’’ contain tables for each class showing the attributes specific to that class. The columns in such a table are:

OM Attribute	lists the attributes specific to the class
Value Syntax	gives the syntax(es) for each value
Value Length	indicates constraints on the number of bits, octets or characters in each value that is a string
Value Number	indicates constraints on the number of values
Value Initially	shows the values *OM-Create*( ) function supplies when initialisation is requested

Below is an example of such a table, taken from XMHS.

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
Confidentiality-Algorithm	Object(Algorithm)	-	0-1	-
Content	Object(Content)	-	1	-
Content-Return-Requested	Boolean	-	1	false
Deferred-Delivery-Time	String(UTC-Time)	0-17	0-1	-
Disclosure-Allowed	Boolean	-	1	false
Latest-Delivery-Time	String(UTC-Time)	0-17	0-1	-
Originator-Return-Address	Object(OR-Name)	-	0-1	-
Priority	Enum(Priority)	-	1	normal

OM Attributes of Message

Recall that an instance of a subclass can be used wherever a particular OM class is required, i.e., the client can supply a subclass as an argument and the service can return a subclass.

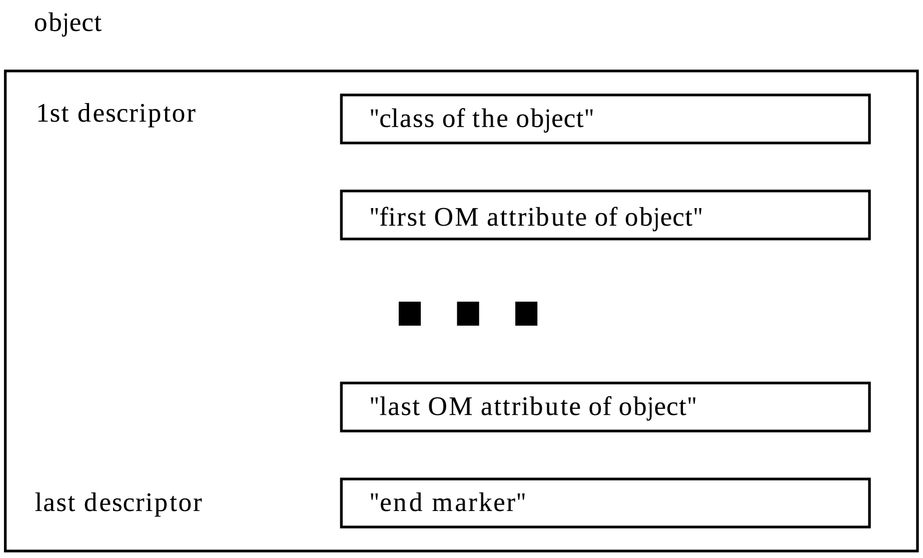
The attributes of an abstract class, through inheritance, are also attributes of the subclasses of the abstract class. In other words, the attributes of an abstract class are those common to all its subclasses.

For example, a message to be submitted to the MTS by the UA is termed the *Submitted-Message*, and since *Submitted-Message* is a subclass of *Submitted-Communique*, the attributes of the latter also apply.

A public object is represented by a list of descriptors where the following apply:

- If present in the representation of an object, the class of the object is generally denoted by the first descriptor.
- To indicate whether the public object is service-generated or client-generated, the syntax field of its first descriptor has the OM_S_SERVICE_GENERATED bit set or cleared as appropriate.
- Each OM attribute of the object is represented by a descriptor. Such descriptors occur before the last descriptor.
- The last descriptor signals the end with its Type component being *no-more-types*, its Syntax component being *no-more-syntaxes* and its Value component being entirely unspecified (refer to XOM - see **Referenced Documents**.)

The following depicts a possible representation of a public object.



Consider the following example, which shows how an object of class **IA5-Text-Body-Part** may be represented as a public, client-generated object in a C program.

The **IA5-Text-Body-Part** class is defined with the following attributes (XMHS):

OM Attribute	Value Syntax	Length	Value Number	Value Initially
Repertoire	Enum(IA5-Repertoire)	-	1	IA5
Text	String(IA5)	-	1	-

OM Attributes of IA5-Text-Body-Part

```
static OM_descriptor some_body_part[] = {
       {OM_CLASS, OM_S_OBJECT_IDENTIFIER_STRING,
              IM_C_IA5_BODY_PART,
              { (OM_string_length) (sizeof(OMP_O_IM_C_IA5_BODY_PART)-1),
                 OMP_O_IM_C_IA5_BODY_PART }},
       {IM_REPERTOIRE,  OM_S_ENUMERATION, IM_IA5 },
       {IM_TEXT,  OM_S_IA5_STRING,
              { (OM_string_length) (sizeof("** data **")-1),
                 "** data **" }},
       OM_NULL_DESCRIPTOR
};
```

The data structure, OM_descriptor, is defined in the **<xom.h>** header in XOM thus:

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

The object ‘‘some_body_part’’ is represented as an array of descriptors. Each descriptor is a triplet of type, syntax and value. Its first descriptor identifies the class of this object. This is followed by the descriptors representing attributes of an object of this class. In this case, the class has single-valued attributes, namely, *Repertoire* and *Text*. OM_NULL_DESCRIPTOR is a macro (defined in XOM) that marks the end of the array of descriptors representing this object.

The attributes of an object are unordered from the client's point of view. However, the values of the attributes are ordered; the position of the first value is zero and positions of successive values are successive positive integers.

Some arguments for the interface functions are of type Object. These may be either public or private objects. The following point out some examples.

- Private objects (whose storage is allocated through the *OM-Create*( ) function) can be built using a combination of OM interface functions, e.g., *OM-Create*( ), *OM-Copy*( ), *OM-Copy-Value*( ), *OM-Put*( ) and *OM-Write*( ). Refer to the MA Submit programming example (**Section 6.6**).
- Client-generated public objects can be built by the client without the need of any OM interface functions. The attribute values of these objects can be initialised or assigned by the client. See the programming example using XDS in XDS (see **Referenced Documents**).

Typically, writing a client program that uses the APIs involve these general steps.

1. Select the Interface Functions to be used.
2. Fill in the arguments for the function calls.

   Arguments of type OM_object may be either public (whose storage is allocated by the client program) or private (whose storage is allocated through the *OM-Create*( ) function).

   a. To check for superclasses for inheritance of attributes, look up the Class Hierarchy section in the relevant API specification document.

b. To know the format for building objects to be used as arguments, look up the Class Definitions in the relevant API specification document.

c. To find the correct symbolic constants of the relevant packages, look up Declaration Summary in the relevant API specification document.

6.5 OVERVIEW OF THE EXAMPLES

The next few sections present some sample programs intended to demonstrate the use of selected MA, MT and OM interface functions. These simple examples do not implement a UA or a gateway. The programming examples are referred to thus:

MA submit example illustrates the use of the MA interface to submit a message

MA deliver example illustrates the use of the MA interface to deliver a message

MT send example illustrates the use of the MT interface to transfer a message out

MT receive example illustrates the use of the MT interface to transfer a message in.

Each example lists the MA or MT interface functions it uses.

In the first two examples, the MA interface is initialised with the following features:

Basic Access FU
Submission FU
Delivery FU
MH 84 Package
IM 84 Package

In the last two examples, the MT interface is initialised with the following features:

Basic Transfer FU
Transfer Out FU
Transfer In FU
MH 84 Package
IM 84 Package

6.5.1 Assumptions

Primarily intended to illustrate the use of the APIs, the sample programs are kept simple and the code given is *not* complete. The sample programs are written in C and have been compiled; however, they are not guaranteed to be correct.

The input facility used in the examples is primitive and error checking is not exhaustive. Extensive error handling and remedial actions are omitted.

6.5.2 Files Used by the Examples

Each example uses different C source files. Refer to the comments at the beginning of the file with the main() program for each example for a list of the files required by the example.

6.5.3 Presentation of the Files

The files are presented thus:

- Files specific for a particular example are presented in the subsection for that example.
- More general purpose files that are used by more than one example are given in the last subsection.

Hence, the files are given under the subsections in the following order:

Subsection	**Files**
MA submit example	ex_MASmain.c ex_MAmakemsg.c ex_MAmakeP1.c ex_MAopen.c (also used in MA deliver example)
MA deliver example	ex_MADmain.c ex_MAgetmsg.c
MT send example	ex_MTSmain.c ex_MTmakemsg.c ex_MTmakeP1.c ex_MTopen.c (also used in MT receive example)
MT receive example	ex_MTRmain.c ex_MTgetmsg.c
Other files (alphabetically)	ex_export.c ex_im_export.h ex_im_import.h ex_import.h ex_makeP2.c ex_read.c ex_utils.c ex_write.c example.h

6.5.4 Compatibility with ISO C

Note that in the C programming examples in this chapter, compatibility with ISO C has not been verified.

6.6 MA SUBMIT EXAMPLE

The objective of this sample client application program is to compose a simple interpersonal message and submit it to the X.400 Message Handling System via the MA interface.

The main steps in this sample program are as follows:

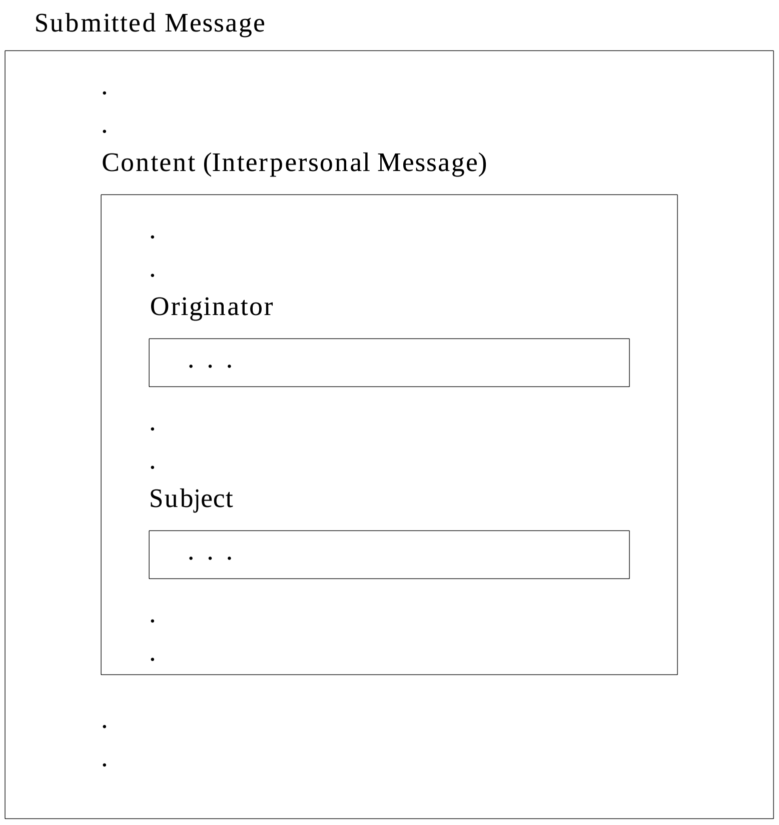
- get user input for establishing MA session
- establish MA session with the service and obtain a workspace
- get user input for message to be composed
- build the message
- release storage for objects
- submit the message
- terminate MA session

A message to be submitted to the MTS by the UA is the **Submitted-Message** (refer to XMHS - see **Referenced Documents**). Since **Submitted-Message** is a subclass of **Submitted-Communique**, refer to XMHS (see **Referenced Documents**) for other relevant (inherited) attributes.

One of the attributes of a **Submitted-Message** is **Content** (refer to XMHS - see **Referenced Documents**). Our example deals with Interpersonal Messaging; hence, **Content** would be expanded using **Interpersonal-Message**, found in XMHS (see **Referenced Documents**).

Our example uses the OM interface functions *OM-Create*( ) and *OM-Put*( ) to build **Submitted-Message** as a private object. Refer to the ex_MAmakemsg.c file, in **Section 6.6.2**.

Submitted-Message can be thought of as an object that comprises several nested, component objects. For illustration, we only show how to build selected, individual components of the **Submitted-Message** (e.g., the **Originator** component of the **Content** component of the **Submitted-Message** object). The other component objects may be built and inserted into the **Submitted-Message** object in a similar fashion.



Values for relevant attributes of interest are to be supplied when constructing the message using the API, and the service implementation has to supply missing values or reject the function call.

6.6.1 ex_MAsmain.c

```
/*
 * File:                ex_MAsmain.c
 *
 * contains:            main( ) for MA submit example
 */

/*
 * This example illustrates the use of the OM and MA interface
 * functions to submit a message.
 * It builds a Submitted Message object for the MA Submit process.
 *
 * This example uses the following MA interface functions:
 *      ma_open( )
 *      ma_submit( )
 *      ma_close( )
 *
 * This example requires these files:
 *      ex_MAsmain.c, ex_MAmakemsg.c, ex_MAmakeP1.c, ex_makeP2.c
 * as well as:
 *      ex_MAopen.c, ex_write.c,
 *      ex_export.c, ex_utils.c
 *      example.h, ex_im_export.h, ex_import.h, ex_im_import.h,
 * and all the header files as defined in the API specification.
 *
 * In order to use this program, the user might need to configure
 * certain variables (e.g., client name, recipient names),
 * or to modify the program appropriately.
 *
 * NOTE: While this example has been compiled successfully,
 *      it has not be verified with an MA interface implementation.
 */

/*
 * NOTE: The following example is not complete.
 */
#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"

                        /* Global variables */
extern OM_private_object        session;        /* the session */
extern OM_private_object        p1_object;      /* the message object */
extern OM_workspace             x4_workspace;   /* the workspace */
```

```
main( )   /* main routine for the MA submit example */
{
  int                     rc = 0;
  int                     i = 0;
  char                    client[64];
  OM_private_object       result; /* submission result */
  OM_object               result_public;
  OM_public_object        id_object;
  OM_value_position       total;
                          /* Message Attributes */
  int                     priority;
  char                    originator[254];
  char                    msgid[254];
  char                    *recipients[11];  /* maximum 10 recipients
                                             * for this example */
  char                    subject[254];
  bodypart                body[2];

 /*
  * Open a session
  *   x4_MAopen( ) uses the MA interface function, ma_open( ), which
  *   return values for "session" and "x4_workspace".
  */
  strcpy(client, CLIENT_NAME);
  printf("Opening session, client is \"%s\"\n", client);
  rc = x4_MAopen(client);
  printf("x4_MAopen( ) returned -> %d\n",rc);

 /*
  * Input values for message building
  */
  printf("\n\nNOTE:  OR_NAMES are actually hardcoded   :NOTE");
  printf("\nNOTE:  in this example.                    :NOTE\n\n");

  printf("\nEnter Priority > ");        fflush(stdout);
  scanf("%d", &priority);

  printf("\nEnter Content ID > ");      fflush(stdout);
  scanf("%s", msgid);

  printf("Enter Originator > ");        fflush(stdout);
  scanf("%s", originator);

  for (i = 0; i < 10; ++ i)
  {
    printf("Enter recipient %d (\".\" to stop) > ", i);  fflush(stdout);
    recipients[i] = (char *)malloc(254);
    scanf("%s", recipients[i]);
    recipients[i+1] = NULL;
    if (recipients[i][0] == '.')
    {
      free(recipients[i]);
```

```
        recipients[i] = NULL;
        i = 11;
      }
    }
    fflush(stdin);

    printf("Enter Subject > ");              fflush(stdout);
    gets(subject);

    printf("Enter Bodypart type > ");        fflush(stdout);
    scanf("%d", &body->bodypart_type);
    printf("Enter Bodypart file > ");        fflush(stdout);
    body[0].bodypart_file = (char *)malloc(254);
    scanf("%s", body[0].bodypart_file);
    body[1].bodypart_type = END_BP;

   /*
    * Print values that message will be built with
    */
    printf("\nPriority: %d\n", priority);
    printf("Originator: %s\n", originator);
    i = 0;
    while ((i < 10) && (recipients[i] != NULL))
    {
          printf("Recipient %d : %s\n", i, recipients[i]);
          ++i;
    }
    printf("Subject: %s\n", subject);
    printf("Bodypart type: %d\n", body[0].bodypart_type);
    printf("Bodypart file: %s\n\n", body[0].bodypart_file);

   /*
    * Build the message
    */
    printf("Building message\n");
    rc = MAmake_msg(priority, msgid, originator, recipients,
                    subject, body );
    printf("MAmake_msg( ) returned -> %d\n",rc);

   /*
    * Submit the message
    */
    printf("sending message\n");
    rc = ma_submit(session, p1_object, &result);
    printf("ma_submit( ) returned -> %d\n",rc);

   /* Get the returned message identifier from the result */
    rc = om_get(result, 0, 0, 0, 0, 0, &result_public, &total);

   /* Search for the MTS Identifier */
```

```
    for (i=0; result[i].type != OM_NO_MORE_TYPES; i++)
    {
          if (result[i].type == MH_T_MTS_IDENTIFIER) break;
    }

    if (result[i].type == MH_T_MTS_IDENTIFIER)
    {
      id_object = (OM_public_object) result[i].value.object.object;
      for (i=0; id_object[i].type != OM_NO_MORE_TYPES; i++)
          if (id_object[i].type == MH_T_LOCAL_IDENTIFIER) break;
      printf("Message Identifier: %s\n", id_object[i].value.string.length);
    }
    else
      printf("Error: No Message Identifier returned\n");

   /*
    * Close session
    */
    rc = ma_close(session);
    printf("ma_close( ) returned -> %d\n",rc);
  }
```

6.6.2 ex_MAmakemsg.c

```
/*
 * File:            ex_MAmakemsg.c
 *
 * contains:        MAmake_msg( )
 */

/*
 * NOTE: The following example is not complete.
 */
#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"
/*
 * Import the object identifiers of OM classes of OM_string type
 * to be used in this file. These object identifiers have been
 * defined as global variables and exported in the file, ex_export.c,
 * by compiling it and linking it with the other code.
 * For convenience, all object identifiers are defined as global
 * variables and imported. The client might choose to define only
 * a subset and to import only those object identifiers required.
 */
#include "ex_import.h"

OM_private_object           p1_object;
extern OM_workspace         x4_workspace;

/*
 * MAmake_msg( )
 *
 *    constructs a message for Submission; this message consists of
 *    a P1 message envelope with values and P2 contents (by calling
 *    makeP2( ) to get the content to be inserted).
 *
 *    For the MA interface, the object to build is of the class,
 *    Submitted Message, which has its own class-specific attributes
 *    as well as  those inherited attributes from its superclass,
 *    Submitted Communique class,
 *
 *    This example uses the minimally required attributes, namely:
 *    (a) Submitted Communique class:
 *            Content Type           (defaulted to P2-1984)
 *            Originator Name        (OR-Name object)
 *            Recipient Descriptor   (RD object)
 *
 *    (b) Submitted Message class:
 *            Content                (Content object; in this example,
 *                                   taken as an Interpersonal Message)
 *     Priority               (for this example, as given in input;
 *                            else, default is "normal")
 * INPUT:
 *    - priority, msgid, originator, recipients, subject, body
 * OUTPUT:
 *    - (p1_object - built up with values from input arguments above)
 * CALLS:
 *    - make_p2( ), MAput_recip_p1( ), str2obj( )
 * RESULT:
 *    - success or error
 */
OM_return_code
MAmake_msg( int            priority,
          char           *msgid,
          char           *originator,
          char           *recipients[],
          char           *subject,
          bodypart       *body)
{
    OM_return_code        str2obj(char *, OM_descriptor **);
    OM_descriptor         p1[10];     /* enough for envelope information
                           * of interest */
    OM_descriptor         recip_desc[4];
    OM_descriptor         list[3];
    OM_object             p2;
    OM_object             address_object;
    OM_descriptor         *address_descriptors;
    OM_object             message_rd_object;
    OM_type               tlist[] = {OM_NO_MORE_TYPES};
    char                  *address_string;
    OM_return_code        status;
    int                   i = 1;
    int                   content_type = MH_CTI_P2_1984;

    if ((status = om_create(MH_C_SUBMITTED_MESSAGE, OM_TRUE,
                x4_workspace, &p1_object)) != OM_SUCCESS)
        return(status);

    MAKECLASS(p1, OMP_O_MH_C_SUBMITTED_MESSAGE);

   /* Attribute: message priority */
    if (priority)
    {
        put_desc(&p1[i++], MH_T_PRIORITY, OM_S_ENUMERATION,
                &priority, 0);
    }

    /* Attribute: message originator */
    if (originator)
    {
        if ((status = str2obj(originator, &address_descriptors))
                    != OM_SUCCESS)
        {
            /* clean up and return */
            return(status);
        }
        put_desc(&p1[i++], MH_T_ORIGINATOR_NAME, OM_S_OBJECT,
                        &address_descriptors, 0);
    }
    else    /* The mandatory field, Originator, is missing. */
    {
        return(ERR_REQ_FIELD_MISSING);
    }

    /* Attribute: (P2) content */
    if ((status = make_p2(msgid, originator, subject, body, &p2))
                        != OM_SUCCESS)
    {
        /* clean up and return */
        return(status);
    }
    put_desc(&p1[i++], MH_T_CONTENT, OM_S_OBJECT, &p2, 0);

    /* Attribute: Content Type */
    put_desc(&p1[i++], MH_T_CONTENT_TYPE, OM_S_INTEGER,
                        &content_type, 0);

    ENDOBJ(p1[i]);

    /* put in these values first before inserting recipients */
    if ((status = om_put(p1_object, OM_REPLACE_ALL, p1,
                         tlist,
                         (OM_value_position) 0,
                         (OM_value_position) 0)) != OM_SUCCESS)
    {
        /* clean up and return error */
        return(status);
    }

    /*
     * Release any private or public objects no longer needed as
     * om_put( ) does not release them but only makes a copy.
     */
    om_delete(p2);
    free(address_descriptors);  /* client generated */

    /* Attribute: message recipients */

    /*
     * The message recipients are represented as an object of
     * the Submitted Message RD class. Such an object includes
     * P1 option attributes and OR-Name as a subobject.
     *
     * Call str2obj( ) to convert each recipient's address string with
     * options into an array of descriptors in the form acceptable as
     * a public object and need not specify any particular class in
```

```
     * this example.
     * Use the "INCLUDE_TYPES" feature of om_put( ) to select the
     * attributes whose values are to be inserted into the OR-Name
     * subobject and into the Submitted Message RD object.
     */

    i = 0;
    while((address_string = recipients[i++]) != NULL)
    {
        if ((status = str2obj(address_string, &address_descriptors))
                        != OM_SUCCESS)
        {
            /* clean up and return */
            return(status);
        }
        if ((status = MAput_recip_p1(address_descriptors,
                                &message_rd_object)) != OM_SUCCESS)
        {
            /* clean up and return */
            return(status);
        }

       /*
        * We could have collected all recipients and
        * perform om_put( ) at once. However, for simplicity,
        * the recipients is inserted one by one.
        */
        MAKECLASS(list, OMP_O_MH_C_SUBMITTED_MESSAGE_RD  );

        put_desc(&list[1], MH_T_RECIPIENT_DESCRIPTORS,
                    OM_S_OBJECT, &message_rd_object, 0);
        ENDOBJ(list[2]);

        if ((status = om_put(p1_object, OM_INSERT_AT_END,
                            list, 0, 0, 0)) != OM_SUCCESS)
        {   /* clean up and return */
            return(status);
        }
    } /* while */

    /* p1_object already has the new object */
    return(OM_SUCCESS);
}
```

6.6.3 ex_MAmakeP1.c

```
/*
 * File:             ex_MAmakeP1.c
 *
 * contains:         MAput_recip_p1( )
 *
 * NOTE: The following example is not complete.
 */
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"
#include "ex_import.h"

extern OM_private_object    session;
extern OM_workspace         x4_workspace;
extern OM_private_object    p1_object;

/*
 * The OR address attribute types of interest to be included in an OR
 * Address object are defined here. These are to be used with om_put( ).
 */
OM_type     address_types[] = {
             MH_T_ADMD_NAME,
             MH_T_COMMON_NAME,
             MH_T_COUNTRY_NAME,
             MH_T_DOMAIN_TYPE_1,
             MH_T_DOMAIN_VALUE_1,
             MH_T_DOMAIN_TYPE_2,
             MH_T_DOMAIN_VALUE_2,
             MH_T_DOMAIN_TYPE_3,
             MH_T_DOMAIN_VALUE_3,
             MH_T_DOMAIN_TYPE_4,
             MH_T_DOMAIN_VALUE_4,
             MH_T_GENERATION,
             MH_T_GIVEN_NAME,
             MH_T_INITIALS,
             MH_T_ORGANIZATION_NAME,
             MH_T_ORGANIZATIONAL_UNIT_NAME_1,
             MH_T_ORGANIZATIONAL_UNIT_NAME_2,
             MH_T_ORGANIZATIONAL_UNIT_NAME_3,
             MH_T_ORGANIZATIONAL_UNIT_NAME_4,
             MH_T_PRMD_NAME,
             MH_T_SURNAME,
             OM_NO_MORE_TYPES
};

/*
 * MAput_recip_p1( )
 *
 *   Makes and set values to attributes in the Submitted Message RD
 *   (Recipient Descriptor) OM object.
 *   Instead of inserting one attribute in at a time, we chose to
 *   form a client public object with all values given and call
 *   om_put( ) once.
 *
 * INPUT:
 *   value   - contains all OR-Address attributes and options
 * OUTPUT:
 *   p1recip - returns a handle to a (private) Submitted Message RD object
 *              (The caller needs to insert this private object into
 *              the P1 message.)
 * RESULT:
 *   - success or error
 */
OM_return_code
MAput_recip_p1(OM_descriptor        *value,
             OM_object              *p1recip)
{
    OM_object              address;
    OM_private_object      recip;
    OM_descriptor          msg_rd[5]; /* enough for this example */
    OM_return_code         status;

   /*
    * Create and build an OR Address object.
    * We have chosen to create a private object even though
    * we could have created a public object.
    *
    * Use default values for all attributes unless
    * overriding values are supplied in input argument, "value".
    */
    if ((status = om_create(MH_C_OR_ADDRESS, OM_TRUE, x4_workspace,
                &address)) != OM_SUCCESS)
        return(status);
    /*
     * Select only the OR Address attribute in "value" identified in
     * "address_types" and insert this into the "address" object.
     */
    if ((status = om_put(address, OM_REPLACE_ALL, value,
                         address_types, 0, 0)) != OM_SUCCESS)
    {
        om_delete(address);
        return(status);
    }

    /*
     * OR Address is a sub-object of Submitted Message RD.
     * This is the only attribute needed.
     */
    if ((status = om_create(MH_C_SUBMITTED_MESSAGE_RD, OM_TRUE,
                            x4_workspace, &recip)) != OM_SUCCESS)
    {   /* clean up and return */
        return(status);
    }
    MAKECLASS(msg_rd, OMP_O_MH_C_SUBMITTED_MESSAGE_RD);

    msg_rd[1].type = MH_T_RECIPIENT_NAME;
    msg_rd[1].syntax = OM_S_OBJECT;
    msg_rd[1].value.object.object = (OM_object) address;
    ENDOBJ(msg_rd[2]);

    if ((status = om_put(recip, OM_INSERT_AT_END, msg_rd, 0, 0, 0))
                                                      != OM_SUCCESS)
    {
        return(status);
    }

    *p1recip = recip;

    return(OM_SUCCESS);
}
```

6.6.4 ex_MAopen.c

```
/*
 * File:            ex_open.c
 *
 * contains:        x4_MAopen( )
 *
 * Note: This example is not complete.
 */
#include <string.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include "example.h"

/*
 * Define and export object identifiers of OM_string type for use in
 * this file as well as in other files.
 */
OM_EXPORT(MH_FE_MH_84)
OM_EXPORT(MH_FE_MH_88)
OM_EXPORT(MH_FE_IM_84)
OM_EXPORT(MH_FE_IM_88)
OM_EXPORT(MH_FE_BASIC_TRANSFER)
OM_EXPORT(MH_FE_BASIC_ACCESS)
OM_EXPORT(MH_FE_TRANSFER_IN)
OM_EXPORT(MH_FE_TRANSFER_OUT)
OM_EXPORT(MH_FE_SUBMISSION)
OM_EXPORT(MH_FE_DELIVERY)

OM_private_object           session;
OM_workspace                x4_workspace;

/*
 * x4_MAopen( )
 *       calls the MA interface function, ma_open( ), to establish
 *       an MA session for a particular client. The MA interface is
 *       is initialised with features are predefined thus:
 *       packages:
 *               MH 84 Package, IM 84 Package
 *       and MA functional units:
 *               Basic Access FU, Submission FU and Delivery FU
 * INPUT:
 *       client        - client’s name
 * OUTPUT:
 *       (global variables: session, x4_workspace get values)
 * CALLS:
 *       ma_open( )
 * RETURNS:
 *        - success or error
 */
```

```
int x4_MAopen(char *client)
{
    MH_feature          feature_list[6];
    OM_string           client_name;
    OM_return_code      status;
    int                 i;

    /*
     * Make sure we can receive and send messages that are 1984 P1/P2
     */
    feature_list[0].feature = MH_FE_BASIC_ACCESS;
    feature_list[1].feature = MH_FE_SUBMISSION;
    feature_list[2].feature = MH_FE_DELIVERY;
    feature_list[3].feature = MH_FE_MH_84;
    feature_list[4].feature = MH_FE_IM_84;
    feature_list[5].feature.length = 0;

    client_name.length = strlen(client);
    client_name.elements = client;

    if ((status = (ma_open( (OM_object) 0,
                            client_name,
                            feature_list,
                            &session,
                            &x4_workspace))) != OM_SUCCESS)
        return(status);

    /* Check for features activated as requested. */
    for (i = 0; feature_list[i].feature.length != 0; i++)
    {
        if (feature_list[i].activated == OM_FALSE)
            return(MH_RC_FEATURE_UNAVAILABLE);
    }
    /* ... etc ... Insert code to check for the other features */

    return(OM_SUCCESS);
}
```

6.7 MA DELIVER EXAMPLE

6.7.1 ex_MAdmain.c

```
/*
 * File:                ex_MAdmain.c
 *
 * contains:            main( ) for MA deliver example
 *
 * This example illustrates the use of the OM and MA interface functions
 * to deliver a message. A message object is obtained from the delivery
 * queue and selected information of the message is read.
 *
 * This example uses the following MA interface functions:
 *       ma_open( )
 *       ma_start_delivery( )
 *       ma_finish_delivery( )
 *       ma_close( )
 *
 *  This example requires these files:
 *      ex_MAdmain.c, ex_MAgetmsg.c
 *  as well as:
 *      ex_MAopen.c,  ex_read.c,
 *      ex_export.c,  ex_utils.c
 *      example.h, ex_im_export.h, ex_import.h, ex_im_import.h,
 *  and all the header files as defined in the API specification.
 *
 *  In order to use this program, the user might need to configure
 *  certain variables (e.g., client name, recipient names),
 *  or to modify the program appropriately.
 *
 *  NOTE:     While this example has been compiled successfully,
 *            it has not be verified with an MA interface
 *            implementation and is not guaranteed to work.
 *            The following example is not complete.
 */
#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"

OM_private_object                p1_object;  /* the message object */
extern OM_private_object         session;
extern OM_workspace              x4_workspace;
```

```
main( )    /* main routine for MA Deliver Example */
{
  int                rc = 0;
  int                i = 0;
  char               client[64];
                     /* Message Attributes */
  int                priority;
  char               msgid[254];
  char               originator[254];
  char               *recipients[11];  /* maximum 10 recipients
                                        * for this example */
  char               subject[254];
  bodypart           body[2];

 /*
  * Open session
  */
  strcpy(client, CLIENT_NAME);
  printf("Opening session, client is \"%s\"\n", client);
  rc = x4_MAopen(client);
  printf("x4_MAopen( ) returned -> %d\n",rc);

 /*
  * Obtain the first message in the input queue
  */
  printf("getting a message\n");
  rc = ma_start_delivery(session, &p1_object);
  printf("ma_start_delivery( ) returned -> %d\n", rc);

 /*
  * Get message identifier, originator, priority, recipients,
  * subject and bodyparts
  */
  printf("Getting information from the message.\n");
  rc = MAgetmsg(&priority, msgid, originator, recipients,
                          subject, body);
  printf("MAgetmsg( ) returned -> %d\n", rc);

 /*
  * Print selected information read from the message.
  */
  printf("Priority: %d\n", priority);
  printf("Message Identifier: %s\n", msgid);
  printf("Originator: %s\n", originator);
  printf("Subject: %s\n", subject);
  printf("BodyPart filename: %s\n", body[0].bodypart_file);
  printf("First Recipient Name: %s\n", recipients[0]);

 /*
  * Complete processing and delete the current message from queue.
  */
  printf("\nFinishing with the message\n");
  rc = ma_finish_delivery(session, (OM_object) 0, (OM_object) 0);
```

```
    printf("ma_finish_delivery( ) returned -> %d\n",rc);

   /*
    * Close session
    */
    rc = ma_close(session);
    printf("ma_close( ) returned -> %d\n", rc);
  }
```

6.7.2 ex_MAgetmsg.c

```
/*
 * File:                ex_MAgetmsg.c
 *
 * contains:            MAgetmsg.c
 */

#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"
#include "ex_import.h"

extern OM_private_object    p1_object;

/*
 * MAgetmsg( )
 *   obtains selected attributes of interest from a Delivered Message
 *   object and returns them to caller
 * INPUT:
 *   priority    - priority of the message
 *   msgid       - the message identifier in the envelope
 *   originator  - the OR address of the originator in the envelope.
 *   recipients  - pointer array to recipient address string. getmsg( )
 *                 allocates memory for the string.
 *   subject     - the subject of the message in P2
 *   body        - pointer to array of bodypart struct that contains
 *                 the type of bodypart and the name of the file that
 *                 holds the data for bodypart
 * OUTPUT:
 *   the input arguments are filled in with data if any.
 * RESULT:
 *   success or error
 */
```

```
OM_return_code
MAgetmsg(      int          *priority,
               char         *msgid,
               char         *originator,
               char         *recipients[],
               char         *subject,
               bodypart     *body)
{
    OM_return_code          obj2str( );
    OM_public_object        public;
    OM_private_object       env_object;
    OM_private_object       p2_object;
    OM_exclusions           exclusions;
    OM_type                 p1_types[]={OM_CLASS,
                                        MH_T_MTS_IDENTIFIER,
                                        MH_T_PRIORITY,
                                        MH_T_ORIGINATOR_NAME,
                                        MH_T_RECIPIENT_DESCRIPTORS,
                                        OM_NO_MORE_TYPES
                                        };
    OM_type                 p2_types[]={OM_CLASS,
                                        IM_SUBJECT,
                                        IM_BODY,
                                        OM_NO_MORE_TYPES};
    OM_public_object       attributes;
    OM_object              subobject;
    OM_value               value;
    OM_value_position      total;
    char                   *address_string;
    OM_return_code         status;
    int                    i, j,
                           recip_cnt = 0;      /* recipient counter */
    int                    bp_cnt = 0;         /* bodypart counter */

    /* Delivered Message object has two key attributes:
     * Content and Envelope
     */
    if ((status = om_get(p1_object, OM_EXCLUDE_SUBOBJECTS, 0, 0, 0, 0,
                                   &public, &total)) != OM_SUCCESS)
        return(status);

    /* Check if object is of class Delivered Message
     * Alternatively, use om_instance( ).
     */
    if (memcmp(MH_C_DELIV_MESSAGE.elements,
               public[0].value.string.elements,
               MH_C_DELIV_MESSAGE.length) != 0)
    {
        om_delete(public);
        return(ERR_NOT_EXPECTED_OBJECT);
    }
```

```
        for (i=0; public[i].type != OM_NO_MORE_TYPES; i++)
        {
            switch(public[i].type) {
                case MH_T_CONTENT:
                    p2_object = public[i].value.object.object;
                    break;
                case MH_T_ENVELOPES:
                    env_object = public[i].value.object.object;
                    break;
            }
        }
        om_delete(public);

       /* Get P1 envelope (Delivery Envelope Object) attribute of interest
        */
        exclusions = OM_EXCLUDE_ALL_BUT_THESE_TYPES;
        if ((status = om_get(env_object, exclusions, p1_types,
                             OM_FALSE, 0, 0, &attributes, &total))
                                  != OM_SUCCESS)
           return(status);

        for (i=1; attributes[i].type != OM_NO_MORE_TYPES; i++)
        {
            value = attributes[i].value;
            switch (attributes[i].type)
            {
                case MH_T_MTS_IDENTIFIER:       /* this is a subobject */
                    subobject = (OM_object) value.object.object;
                    j = 0;
                    while (subobject[++j].type != MH_T_LOCAL_IDENTIFIER)
                        /* search for the Local Identifier type,
                         * skipping other attribute types
                         */
                        ;
                    strncpy(msgid, subobject[j].value.string.elements,
                            subobject[j].value.string.length);
                    msgid[subobject[j].value.string.length] = ' ';
                    break;

                case MH_T_PRIORITY:
                    *priority = value.integer;
                    break;

                case MH_T_ORIGINATOR_NAME:
                    /* this is a OR-Name subobject */
                    subobject = (OM_object) value.object.object;
                    obj2str(subobject, originator);
                    break;

                case MH_T_RECIPIENT_DESCRIPTORS:
                    /* this is a suboject of which ORName/ORAddress
                     * is a subobject.
                     */
                    subobject = (OM_object) value.object.object;

                    j = 0;
                    while (subobject[++j].type != MH_T_RECIPIENT_NAME)
                        /* search for the Recipient Name type,
                         * skipping other attribute types
                         */
                        ;
                    if ((recipients[recip_cnt] = (char *) malloc(256))
                                                      == NULL)
                        return(OM_MEMORY_INSUFFICIENT);

                    obj2str(subobject[j].value.object.object,
                            recipients[recip_cnt++]);
                    break;

                default:
                    break;
            } /* switch */
        } /* for */

        /* Release this service-generated public object, no longer needed */
        om_delete(attributes);

       /*
        * Obtain P2 (Interpersonal Message Object) attributes of interest
        */
        exclusions = OM_EXCLUDE_ALL_BUT_THESE_TYPES | OM_EXCLUDE_SUBOBJECTS;
        if ((status = om_get(p2_object, exclusions, p2_types,
                             OM_FALSE, 0, 0, &attributes, &total))
                                                      != OM_SUCCESS)
            return(status);

        if (memcmp(IM_C_INTERPERSONAL_MSG.elements,
                   attributes[0].value.string.elements,
                   IM_C_INTERPERSONAL_MSG.length) != 0)
        {
            om_delete(attributes);
            return(ERR_NOT_EXPECTED_OBJECT);
        }

        for (i=1; attributes[i].type != OM_NO_MORE_TYPES; i++)
        {
            value = attributes[i].value;
            switch (attributes[i].type)
            {
                case IM_SUBJECT:
                    strncpy(subject, value.string.elements,
                                    value.string.length);
                    subject[value.string.length] = ' ';
                    break;
                case IM_BODY:
                    /*
                     * The number of body part should be only 1 in
                     * this example, since body[2] was allocated.
                     */
                    subobject = (OM_private_object) value.object.object;
                    getbody(subobject, &body[bp_cnt++]);
                    break;
            }
        }

        /* Terminate the bodypart structure */
        body[bp_cnt].bodypart_type = END_BP;

        om_delete(attributes);
        return(OM_SUCCESS);
    }
```

6.8 MT SEND EXAMPLE

6.8.1 ex_MTsmain.c

```
/*
 * File:        ex_MTsmain.c
 *
 * contains:    the main program for the MT send example
 *
 * This is a simple example to illustrate the use of
 * the OM interface and MT interface functions to send messages.
 * It get input for selected information to build a message and
 * then transfers this message to the output queue.
 * Note: this is not a complete gateway.
 *
 * This example uses the following MT interface functions:
 *                mt_open( )
 *                mt_transfer_out( )
 *                mt_close( )
 *
 * This example requires these files:
 *     ex_MTsmain.c,  ex_MTmakemsg.c,  ex_MTmakeP1.c,
 * as well as:
 *     ex_MTopen.c,  ex_write.c,
 *     ex_utils.c,  ex_export.c,
 *     example.h,  ex_im_export.h,  ex_import.h,  ex_im_import.h,
 * and all the header files as defined in the API specification.
 *
 * In order to use this program, the user might need to configure
 * certain variables (e.g., client name, recipient names),
 * or to modify the program appropriately.
 *
 * NOTE: While it has been verified by one implementation,
 *       there is no guarantee that it will work with other
 *       implementations.
 *       It is not a complete example and is not guaranteed to be correct.
 */
#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"

extern OM_private_object       session;       /* the session */
extern OM_private_object       p1_object;     /* the message object */
extern OM_workspace            x4_workspace;  /* the workspace */
```

```
main( )
{
  int                       rc = 0;
  int                       i = 0;
  char                      client[64];
                            /* message attributes */
  int                       priority;
  char                      msgid[254];
  char                      originator[254];
  char                      *recipients[11];  /* max 10 recipients for
                                               * this example */
  char                      subject[254];
  bodypart                  body[2];

 /*
  * Get input values for this example to build a message
  */
  printf("\n\nNOTE:  OR_NAMES are actually hardcoded   :NOTE");
  printf("\nNOTE:  in this example.                     :NOTE\n\n");
  printf("\nEnter Priority > "); fflush(stdout);
  scanf("%d", &priority);

  printf("Enter message id > ");  fflush(stdout);
  scanf("%s", msgid);

  printf("Enter originator > ");  fflush(stdout);
  scanf("%s", originator);

  for (i = 0; i < 10; ++ i)
  {
    printf("Enter recipient %d (\".\" to stop) > ", i);
    fflush(stdout);
    recipients[i] = (char *)malloc(254);
    scanf("%s", recipients[i]);
    recipients[i+1] = NULL;
    if (recipients[i][0] == '.')
    {
      free(recipients[i]);
      recipients[i] = NULL;
      i = 11;
    }
  }
  fflush(stdin);

  printf("Enter subject > ");        fflush(stdout);
  gets(subject);

  printf("Enter Bodypart type > ");  fflush(stdout);
  scanf("%d", &body->bodypart_type);
  printf("Enter Bodypart file > ");  fflush(stdout);
  body[0].bodypart_file = (char *)malloc(254);
  scanf("%s", body[0].bodypart_file);
  body[1].bodypart_type = END_BP;
 /*
  * Print values that message will be built with
  */
  printf("\nPriority: %d\n", priority);
  printf("Message ID: %s\n", msgid);
  printf("Originator: %s\n", originator);
  i = 0;
  while ((i < 10) && (recipients[i] != NULL))
  {
    printf("Recipient %d : %s\n", i, recipients[i]);
    ++i;
  }
  printf("Subject: %s\n", subject);
  printf("Bodypart type: %d\n", body[0].bodypart_type);
  printf("Bodypart file: %s\n\n", body[0].bodypart_file);

 /*
  * Open session
  */
  strcpy(client, CLIENT_NAME); /* CLIENT_NAME is defined in example.h */
  printf("Opening session, client is \"%s\"\n", client);
  rc = x4_MTopen(client);
  printf("x4_MTopen( ) returned -> %d\n", rc);

 /*
  * Build the message
  */
  printf("Building message\n");
  rc = MTmake_msg(priority, msgid, originator, recipients,
                subject, body );
  printf("MTmake_msg( ) returned -> %d\n",rc);

 /*
  * Transfer the message out
  */
  printf("Sending message\n");
  rc = mt_transfer_out(session, p1_object);
  printf("mt_transfer_out( ) returned -> %d\n", rc);

 /*
  * Close session
  */
  rc = mt_close(session);
  printf("mt_close( ) returned -> %d\n", rc);
}
```

6.8.2 ex_MTmakemsg.c

```
/*
 * File:          ex_makemsg.c
 *
 * contains:      MTmake_msg( ), make_msgid( )
 *
 * NOTE: The following example is not complete.
 */

#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"
/*
 * Import the object identifiers of OM classes of OM_string type
 * to be used in this file. These object identifiers have been
 * defined as global variables and exported in the file, ex_export.c,
 * by compiling it and linking it with the other code.
 * For convenience, all object identifiers are defined as global
 * variables and imported. The client might choose to define only a
 * subset and to import only those object identifiers required.
 */
#include "ex_import.h"

OM_private_object              p1_object;
extern OM_workspace            x4_workspace;

/*
 * MTmake_msg( )
 *
 *   constructs a message for Transfer Out.
 *   This message consists of a P1 message envelope with values
 *   and P2 contents (by calling makeP2( ) to get the content to
 *   be inserted).
 *   For the MT interface, the object to build is of the class, Message,
 *   which has its own class-specific attributes as well as
 *   those inherited attributes from its superclass, Communique class,
 *
 *   This example uses the minimally required attributes, namely:
 *   (a) Communique class:
 *         Content Type         (defaulted to P2-1984)
 *         MTS Identifier       (defaulted as assigned by the service)
 *         Originator Name      (OR-Name object)
 *         Recipient Descriptor (RD object)
 *
 *   (b) Message class:
 *         Content              (Content object; in this example,
 *                              taken as an Interpersonal Message)
 *         Priority             (for this example, as given in input;
 *                                  else, default is "normal")
 * INPUT:
 *   - priority, msgid, originator, recipients, subject, body
 * OUTPUT:
 *   - (p1_object - built up with values from input arguments above)
 * CALLS:
 *   - make_p2( ), MTput_recip_p1( ), str2obj( )
 * RESULT:
 *   - success or error
 */
OM_return_code
MTmake_msg(   int     priority,
        char          *msgid,
        char          *originator,
        char          *recipients[],
        char          *subject,
        bodypart      *body)
{
    OM_return_code    str2obj(char *, OM_descriptor **);
    OM_return_code    make_msgid(char *, OM_object *);
    OM_descriptor     p1[10]; /* enough for envelope info of interest */
    OM_descriptor     recip_desc[4];
    OM_descriptor     mtsid_desc[5];
    OM_descriptor     list[3];
    OM_object         p2;
    OM_object         msgid_object;
    OM_object         address_object;
    OM_descriptor     *address_descriptors;
    OM_object         message_rd_object;
    char              *address_string;
    OM_return_code    status;
    int               i = 1;
    int               content_type = MH_CTI_P2_1984;

    if ((status = om_create(MH_C_MESSAGE, OM_TRUE,
                x4_workspace, &p1_object)) != OM_SUCCESS)
        return(status);

    MAKECLASS(p1, OMP_O_MH_C_MESSAGE);

    /* message priority   */
    if (priority)
    {
        put_desc(&p1[i++], MH_T_PRIORITY, OM_S_ENUMERATION, &priority, 0);
    }

    /* message originator */
    if (originator)
    {
        if ((status = str2obj(originator, &address_descriptors))
                                                != OM_SUCCESS)
        {
            /* clean up and return */
            return(status);
        }
        put_desc(&p1[i++], MH_T_ORIGINATOR_NAME, OM_S_OBJECT,
                     &address_descriptors, 0);
    }
    else    /∗ The mandatory field, Originator, is missing ∗/
    {
        return(ERR_REQ_FIELD_MISSING);
    }

    /∗ message identifier ∗/
    if (msgid)
    {
        if ((status = make_msgid(msgid, &msgid_object)) != OM_SUCCESS)
        {
            return(status);
        }
        put_desc(&p1[i++], MH_T_MTS_IDENTIFIER, OM_S_OBJECT,
                      &msgid_object, 0);
    }

    /∗ P2 content ∗/
    if ((status = make_p2(msgid, originator, subject, body, &p2))
                                              != OM_SUCCESS)
    {
        /∗ clean up and return ∗/
        return(status);
    }
    put_desc(&p1[i++], MH_T_CONTENT, OM_S_OBJECT, &p2, 0);

    /∗ Content Type ∗/
    put_desc(&p1[i++], MH_T_CONTENT_TYPE, OM_S_INTEGER, &content_type, 0);

    ENDOBJ(p1[i]);

    /∗ Put in these values first before inserting the recipients ∗/
    if ((status = om_put(p1_object, OM_REPLACE_ALL, p1,
                         (OM_type_list) 0, (OM_value_position) 0,
                         (OM_value_position) 0)) != OM_SUCCESS)
    {
        /∗ clean up and return error ∗/
        return(status);
    }

   /∗ Release any private or public objects no longer needed as
    ∗ om_put( ) does not release them but only makes a copy.
    ∗/
    om_delete(p2);
    om_delete(msgid_object);
    free(address_descriptors);  /∗ client generated ∗/
```

```
/* message recipients */

/* P1 recipient takes on Message RD class which has
 * P1 option attributes and ORName as a subobject.
 *
 * Call str2obj( ) to convert each recipient's address string with
 * options into an array of descriptors in the form acceptable as
 * a public object and
 * no need to specify any particular class in this example.
 * Use the "INCLUDE_TYPES" feature of om_put( ) to select the
 * attributes whose values are to be inserted into
 * the OR-Name subobject and into Message RD object.
 *
 * Message RD includes MTA Report Request, MTA Responsibility and
 * Recipient Number. The first two are respectively defaulted to
 * values: non-delivery and "true".
 *
 * Must ensure that the string must include RECIPIENT NUMBER,
 * indicating the ordinal position of a recipient on the list of
 * recipients.
 */

i = 0;
while((address_string = recipients[i++]) != NULL)
{
    if ((status = str2obj(address_string, &address_descriptors))
                            != OM_SUCCESS)
    {
        /* clean up and return */
        return(status);
    }
    if ((status = MTput_recip_p1(address_descriptors,
                             &message_rd_object)) != OM_SUCCESS)
    {
        /* clean up and return */
        return(status);
    }

   /*
    * We could have collected all recipients and
    * perform om_put( ) at once. However, for simplicity,
    * the recipients is inserted one by one.
    */
    MAKECLASS(list, OMP_O_MH_C_MESSAGE_RD );

    put_desc(   &list[1], MH_T_RECIPIENT_DESCRIPTORS,
                OM_S_OBJECT, &message_rd_object, 0);

    ENDOBJ(list[2]);

    if ((status = om_put(p1_object, OM_INSERT_AT_END,
                         list, 0, 0, 0)) != OM_SUCCESS)
    {   /* clean up and return */
```

```
                return(status);
            }
        }

        /* p1_object already has the new object */
        return(OM_SUCCESS);
    }

    /* make_msgid( )
     *
     *          Makes a private MTS IDENTIFIER object as
     *          the Local Identifier attribute using input msgid
     *          as the Local Identifier attribute.
     *          Default values will be used for Country/ADMD/PRMD.
     * INPUT:
     *          msgid               - message identifier string
     * OUTPUT:
     *          msgid_object        - a private MTS IDENTIFIER object.
     * RESULT:
     *          success or error
     */
    OM_return_code
    make_msgid( char        *msg_id,
                OM_object   *msgid_object)

    {
        OM_private_object   id_private;
        OM_descriptor       id_public[3];
        OM_return_code      status;

        if ((status = om_create(MH_C_MTS_IDENTIFIER, OM_TRUE,
                     x4_workspace, &id_private)) != OM_SUCCESS)
            return(status);

        MAKECLASS(id_public, OMP_O_MH_C_MTS_IDENTIFIER);

        put_desc(&id_public[1], MH_T_LOCAL_IDENTIFIER,
                     OM_S_IA5_STRING, msg_id, strlen(msg_id));

        ENDOBJ(id_public[2]);

        if ((status = om_put(id_private, OM_REPLACE_ALL, id_public,
                              0, 0, 0)) != OM_SUCCESS)
        {
            om_delete(id_private);
            return(status);
        }

        *msgid_object = id_private;

        return(OM_SUCCESS);
    }
```

6.8.3 ex_MTmakeP1.c

```
/*
 * File:          ex_makeP1.c
 *
 * contains:      MTput_recip_p1( )
 *
 * NOTE: The following example is not complete.
 */

#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"
#include "ex_import.h"

extern OM_private_object    session;
extern OM_workspace         x4_workspace;
extern OM_private_object    p1_object;

/*
 * The OR address attribute types of interest to be included
 * in an OR-Address object is defined as follows.
 * This will be used in a call to om_put( ).
 */
OM_type     address_types[] = {
             MH_T_ADMD_NAME,
             MH_T_COMMON_NAME,
             MH_T_COUNTRY_NAME,
             MH_T_DOMAIN_TYPE_1,
             MH_T_DOMAIN_VALUE_1,
             MH_T_DOMAIN_TYPE_2,
             MH_T_DOMAIN_VALUE_2,
             MH_T_DOMAIN_TYPE_3,
             MH_T_DOMAIN_VALUE_3,
             MH_T_DOMAIN_TYPE_4,
             MH_T_DOMAIN_VALUE_4,
             MH_T_GENERATION,
             MH_T_GIVEN_NAME,
             MH_T_INITIALS,
             MH_T_ORGANIZATION_NAME,
             MH_T_ORGANIZATIONAL_UNIT_NAME_1,
             MH_T_ORGANIZATIONAL_UNIT_NAME_2,
             MH_T_ORGANIZATIONAL_UNIT_NAME_3,
             MH_T_ORGANIZATIONAL_UNIT_NAME_4,
             MH_T_PRMD_NAME,
             MH_T_SURNAME,
             OM_NO_MORE_TYPES
};
```

```
/*
 * MTput_p1_recip(OM_descriptor_list value, OM_object p1recip)
 *
 *    Makes and set values to attributes in the Message RD
 *    (Recipient Descriptor) OM object.
 *    Instead of inserting one attribute in at a time, we chose to form
 *    a client public object with all values given and call om_put( ) once.
 * INPUT:
 *    value   - contains all OR-Address attributes and options
 * OUTPUT:
 *    p1recip - returns a handle to a (private) Message RD object
 *                 (The caller needs to insert this private object into
 *                 the P1 message.)
 * RESULT:
 *    - success or error
 */
OM_return_code
MTput_recip_p1(     OM_descriptor       *value,
                    OM_object           *p1recip)
{
    OM_object           address;
    OM_private_object   recip;
    OM_descriptor       msg_rd[5];
    OM_type             msg_rd_types[] =
                                   {    MH_T_MTA_REPORT_REQUEST,
                                  MH_T_MTA_RESPONSIBILITY,
                                  MH_T_RECIPIENT_NUMBER,
                                  OM_NO_MORE_TYPES
                        };
    OM_return_code      status;

   /*
    * Create and build an OR-Address object.
    * We have chosen to create a private object even though
    * we could have created a public object.
    *
    * Use default values for all attributes unless
    * overriding values are supplied in input argument, "value".
    */
    if ((status = om_create(MH_C_OR_ADDRESS, OM_TRUE,
                        x4_workspace, &address)) != OM_SUCCESS)
        return(status);

   /*
    * Select only the OR Address attribute in "value" identified in
    * "address_types" and insert this into the "address" object.
    */
    if ((status = om_put(address, OM_REPLACE_ALL, value,
                 address_types, 0, 0)) != OM_SUCCESS)
    {
        om_delete(address);
        return(status);
    }
```

```
        /*
         * First, insert MTA Responsibility,
         * MTA Report Request and
         * Recipient Number, if any.
         *
         * Finally, insert the OR-Address as a subobject of
         * the Message RD object.
         */
        if ((status = om_create(MH_C_MESSAGE_RD, OM_TRUE,
                       x4_workspace, &recip))
                                           != OM_SUCCESS)
        {   /* clean up and return */
            return(status);
        }

        if ((status = om_put(recip, OM_REPLACE_ALL, value,
                               msg_rd_types, 0, 0))
                                       != OM_SUCCESS)
        {   /* clean up and return */
            return(status);
        }

        MAKECLASS(msg_rd, OMP_O_MH_C_MESSAGE_RD);
        msg_rd[1].type = MH_T_RECIPIENT_NAME;
        msg_rd[1].syntax = OM_S_OBJECT;
        msg_rd[1].value.object.object = (OM_object) address;
        ENDOBJ(msg_rd[2]);

        if ((status = om_put(recip, OM_INSERT_AT_END, msg_rd, 0,
                                    0, 0)) != OM_SUCCESS)
        {
            return(status);
        }

        *p1recip = recip;

        return(OM_SUCCESS);
    }
```

6.8.4 ex_MTopen.c

```
/*
 * File:            ex_MTopen.c
 *
 * contains:        x4_MTopen( )
 *
 * Note: The following example is not complete.
 */
#include <string.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include "example.h"

/* Define and export object identifiers of OM_string type
 * that are used in this file and other files.
 */
OM_EXPORT(MH_FE_MH_84)
OM_EXPORT(MH_FE_MH_88)
OM_EXPORT(MH_FE_IM_84)
OM_EXPORT(MH_FE_IM_88)
OM_EXPORT(MH_FE_BASIC_TRANSFER)
OM_EXPORT(MH_FE_TRANSFER_IN)
OM_EXPORT(MH_FE_TRANSFER_OUT)
OM_EXPORT(MH_FE_SUBMISSION)
OM_EXPORT(MH_FE_DELIVERY)

OM_private_object           session;
OM_workspace                x4_workspace;
/*
 * x4_MTopen( )
 *     Opens an MT session for a particular client.
 *
 *     The features are predefined with MH_84 and IM_84 packages,
 *     and Basic Transfer, Transfer In and Transfer Out FUs.
 */
int x4_MTopen(char *client)
{
    MH_feature          feature_list[6];
    OM_string           client_name;
    OM_return_code      status;
    int                 i;
```

```
         /* Make sure we can receive and send messages that are 1984 P1/P2 */
         /*
          * Note: The following structure assignment may not be possible
          * with some ANSI C compilers.
          */
          feature_list[0].feature = MH_FE_BASIC_TRANSFER;
          feature_list[1].feature = MH_FE_TRANSFER_IN;
          feature_list[2].feature = MH_FE_TRANSFER_OUT;
          feature_list[3].feature = MH_FE_MH_84;
          feature_list[4].feature = MH_FE_IM_84;
          feature_list[5].feature.length = 0;

         /* The 1st argument to mt_open( ), client_name, must be
          * in the OM_string structure, in length-unspecified form,
          * i.e., a null-terminated string.
          *
          * This is also used as the 2nd argument, client_instance.
          */
        client_name.length        = OM_LENGTH_UNSPECIFIED;
        client_name.elements      = client;

        if ((status = (mt_open(client_name, client_name,
                               feature_list, &session,
                               &x4_workspace))) != OM_SUCCESS)
            return(status);

        /* Check for features activated as requested. */
        for (i = 0; feature_list[i].feature.length != 0; i++)
        {
            if (feature_list[i].activated == OM_FALSE)
                return(MH_RC_FEATURE_UNAVAILABLE);
        }
        /* etc ... insert code to check for other features requested */

        return(OM_SUCCESS);
    }
```

6.9 MT RECEIVE EXAMPLE

6.9.1 ex_MTrmain.c

```
/*
 * File:        ex_MTrmain.c
 *
 * contains:    main( ) for the MT receive example
 *
 * This is a simple example to illustrate the use of
 * the OM interface and MT interface functions to receive messages.
 * It obtains from the input queue a message object and
 * reads selected information from the message.
 *
 * This example uses the following MT interface functions:
 *               mt_open( )
 *               mt_start_transfer_in( )
 *               mt_finish_transfer_in( )
 *               mt_close( )
 *
 *  This example requires these files:
 *      ex_MTrmain.c,  ex_MTgetmsg.c
 *  as well as:
 *      ex_MTopen.c,  ex_read.c,
 *      ex_utils.c,  ex_export.c,
 *      example.h,  ex_im_export.h,  ex_import.h,  ex_im_import.h,
 *  and all the header files as defined in the API specification.
 *
 *  In order to use this program, the user might need to configure
 *  certain variables (e.g., client name, recipient names),
 *  or to modify the program appropriately.
 *
 *  NOTE:    While this example has been compiled successfully,
 *            it is not guaranteed to be correct.
 */
#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"

OM_private_object             p1_object;  /* the message object */
extern OM_private_object      session;
extern OM_workspace           x4_workspace;
```

```
main( )          /* main routine for MT receive example */
{
  int              rc = 0;
  int              i = 0;
  char             client[64];
                   /* Message Attributes */
  int              priority;
  char             msgid[254];
  char             originator[254];
  char             *recipients[11];  /* maximum 10 recipients
                                      * for this example */
  char             subject[254];
  bodypart         body[2];
                   /* identifier for current object returned
                    * in mt_start_transfer_in( ) */
  char             objid_str[65];
  OM_string        objid={ 0, (void *)&objid_str[0] };

 /*
  * Open session
  */
  strcpy(client, CLIENT_NAME); /* client name defined in example.h */
  printf("Opening session, client is \"%s\"\n", client);
  rc = x4_MTopen(client);
  printf("x4_MTopen( ) returned -> %d\n", rc);

 /*
  * Obtain the first message in the input queue
  */
  printf("getting a message\n");
  rc = mt_start_transfer_in(session, &p1_object, objid);
  printf("mt_start_transfer_in( ) returned -> %d\n", rc);

 /*
  * Get the following message attributes:
  *           message identifier, originator, priority, recipients,
  *           subject, bodyparts
  */
  printf("Getting information from the message.\n");
  rc = MTgetmsg(&priority, msgid, originator, recipients,
                          subject, body);
  printf("MTgetmsg( ) returned -> %d\n", rc);

 /*
  * Print selected message attributes read above.
  */
  printf("Priority: %d\n", priority);
  printf("Message Identifier: %s\n", msgid);
  printf("Originator: %s\n", originator);
  printf("Subject: %s\n", subject);
  printf("BodyPart filename: %s\n", body[0].bodypart_file);
  printf("First Recipient Name: %s\n", recipients[0]);
 /*
  * Complete processing and
  * delete the current message from the input queue.
  */
  printf("\nFinishing with the message\n");
  rc = mt_finish_transfer_in(session, p1_object, OM_TRUE);
  printf("mt_finish_transfer_in( ) returned -> %d\n",rc);

 /*
  * Close session
  */
  rc = mt_close(session);
  printf("mt_close( ) returned -> %d\n",rc);
}
```

6.9.2 ex_MTgetmsg.c

```
/*
 * File:          ex_MTgetmsg.c
 *
 * contains:      MTgetmsg( )
 */
#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"
#include "ex_import.h"

extern OM_private_object            p1_object;

/*
 * MTgetmsg( )
 *   gets the attributes of interest from a Message Object
 *   and returns them to caller.
 *
 * INPUT:
 *   priority    -  priority of the message
 *   msgid       -  the message identifier in the envelope
 *   originator  -  the OR-Address of the originator in the envelope.
 *   recipients  -  pointer array to recipient address string.
 *                  getmsg( ) memory for the string.
 *   subject     -  the subject of the message in P2
 *   body        -  pointer to array of bodypart struct to contain the
 *                  type of bodypart and the bodypart filename.
 * OUTPUT:
 *   The input arguments are filled in with data if any.
 * RESULT:
 *   success or error
 */
OM_return_code
MTgetmsg( int           *priority,
        char            *msgid,
        char            *originator,
        char            *recipients[],
        char            *subject,
        bodypart        *body)
{
    OM_return_code      obj2str( );
    OM_private_object   p2_object;
    OM_exclusions       exclusions;
    OM_type             p1_types[]={OM_CLASS,
                                    MH_T_MTS_IDENTIFIER,
                                    MH_T_PRIORITY,
                                    MH_T_ORIGINATOR_NAME,
                                       MH_T_RECIPIENT_DESCRIPTORS,
                                       OM_NO_MORE_TYPES
                                    };
    OM_type            p2_types[]={OM_CLASS,
                                    IM_SUBJECT,
                                    IM_BODY,
                                    OM_NO_MORE_TYPES};
    OM_public_object    attributes;
    OM_object           subobject;
    OM_value            value;
    OM_value_position   total;
    char                *address_string;
    OM_return_code      status;
    int                 i, j, recip_cnt = 0;
    int                 bp_cnt = 0;

   /* Get P1 envelope (Message Object) attribute of interest */
    exclusions = OM_EXCLUDE_ALL_BUT_THESE_TYPES;
    if ((status = om_get(p1_object, exclusions, p1_types, OM_FALSE,
                         0, 0, &attributes, &total)) != OM_SUCCESS)
        return(status);

   /* Check if object is of class Message or Report or something else
    * The class attribute must be the first attribute.
    * Alternatively, use om_instance( ).
    */
    if (memcmp(MH_C_MESSAGE.elements, attributes[0].value.string.elements,
               MH_C_MESSAGE.length) != 0)
    {
        om_delete(attributes);
        return(ERR_NOT_EXPECTED_OBJECT);
    }

    for (i=1; attributes[i].type != OM_NO_MORE_TYPES; i++)
    {
        value = attributes[i].value;
        switch (attributes[i].type)
        {
            case MH_T_MTS_IDENTIFIER:       /* this is a subobject */
                subobject = (OM_object) value.object.object;
                j=0;
                while (subobject[++j].type != MH_T_LOCAL_IDENTIFIER)
                    /* search for this type,
                     * skipping other attribute types */
                    ;
                strncpy(msgid, subobject[j].value.string.elements,
                       subobject[j].value.string.length);
                msgid[subobject[j].value.string.length] = ' ';
                break;

            case MH_T_PRIORITY:
                *priority = value.integer;
                break;
    case MH_T_ORIGINATOR_NAME:  /* this is a OR Name subobject */
             subobject = (OM_object) value.object.object;
             obj2str(subobject, originator);
             break;

    case MH_T_RECIPIENT_DESCRIPTORS:
            /* this is a suboject; and in turn, one of its
             * subobject is the ORName/ORAddress.
             */
             subobject = (OM_object) value.object.object;
             j = 0;
             while (subobject[++j].type != MH_T_RECIPIENT_NAME)
                    /* search for this type,
                     * skipping other attribute types */
                    ;
             if ((recipients[recip_cnt] = (char *) malloc(256)) == NULL)
                    return(OM_MEMORY_INSUFFICIENT);

             obj2str(subobject[j].value.object.object,
                       recipients[recip_cnt++]);
             break;

         default:
                break;
        } /* switch */
    } /* for */

   /* Release this service-generated public object, no longer needed */
    om_delete(attributes);

   /*
    * Obtain P2 (Interpersonal Message Object) attributes of interest
    */

   /* Get P2 as a private subobject.
    * The first attribute of this public object should be
    * a handle to the P2 (private) subobject.
    */
    p1_types[1] = MH_T_CONTENT; /* p1_types[0] requests OM_CLASS */
    p1_types[2] = OM_NO_MORE_TYPES;
    exclusions = OM_EXCLUDE_ALL_BUT_THESE_TYPES  | OM_EXCLUDE_SUBOBJECTS;
    if ((status = om_get(p1_object, exclusions, p1_types, OM_FALSE,
                                0, 0, &attributes, &total)) != OM_SUCCESS)
        return(status);
    p2_object = (OM_private_object) attributes[1].value.object.object;

   /*
    * Get attributes of interests in P2 and return as public object
    */
    exclusions = OM_EXCLUDE_ALL_BUT_THESE_TYPES  | OM_EXCLUDE_SUBOBJECTS;
    if ((status = om_get(p2_object, exclusions, p2_types, OM_FALSE,
                         0, 0, &attributes, &total))  != OM_SUCCESS)
        return(status);
    if (memcmp(IM_C_INTERPERSONAL_MSG.elements,
               attributes[0].value.string.elements,
               IM_C_INTERPERSONAL_MSG.length) != 0)
    {
        om_delete(attributes);
        return(ERR_NOT_EXPECTED_OBJECT);
    }

    for (i=1; attributes[i].type != OM_NO_MORE_TYPES; i++)
    {
        value = attributes[i].value;
        switch (attributes[i].type)
        {
            case IM_SUBJECT:
                strncpy(subject, value.string.elements,
                                 value.string.length);
                subject[value.string.length] = ' ';
                break;

            case IM_BODY:
               /* The number of body part should be only 1 in
                * this example, since allocate body[2].
                */
                subobject = (OM_private_object) value.object.object;
                getbody(subobject, &body[bp_cnt++]);
                break;
        }
    }

    /* Terminate the bodypart structure body */
    body[bp_cnt].bodypart_type = END_BP;

    om_delete(attributes);

    return(OM_SUCCESS);
}
```

6.10 OTHER FILES

6.10.1 ex_export.c

```
/*
 * File:           ex_export.c
 */

/*
 * This file defines the global variables for all the
 * object identifiers defined in XOM and X.400 API.
 * The names of the variables are those between the parentheses.
 * Each compilation unit needs to import these variables using
 * the macro OM_IMPORT(CLASS_VARIABLE_NAME).
 *
 * For convenience, all object identifiers are defined and
 * exported/imported. However, a particular client might select only
 * those object identifiers for its application.
 *
 * Exported object identifiers are in the file:
 *        ex_export.c
 * which also #includes
 *        ex_im_export.h
 * for exporting object identifiers belonging to the IM package.
 * These have to be compiled.
 *
 * Imported object identifiers are in files:
 *        ex_import.h
 * and, for object identifiers belonging to the IM package, in:
 *        ex_im_import.h
 * These files have to be included in the compilation unit.
 */

#include <xom.h>
#include <xmhp.h>
#include <ximp.h>
#include <xsmp.h>

OM_EXPORT(MH_C_ALGORITHM)
OM_EXPORT(MH_C_ALGORITHM_AND_RESULT)
OM_EXPORT(MH_C_ASYMMETRIC_TOKEN)
OM_EXPORT(MH_C_BILATERAL_INFO)
OM_EXPORT(MH_C_COMMUNIQUE)
OM_EXPORT(MH_C_CONTENT)
OM_EXPORT(MH_C_DELIV_MESSAGE)
OM_EXPORT(MH_C_DELIV_PER_RECIP_DR)
OM_EXPORT(MH_C_DELIV_PER_RECIP_NDR)
OM_EXPORT(MH_C_DELIV_PER_RECIP_REP)
OM_EXPORT(MH_C_DELIV_REPORT)
```

```
/*
 * NOTE: the following two names are not in alphabetical order
 * to retain the value assignment as published in X.400 API Version 2.0
 */
OM_EXPORT(MH_C_DELIVERY_CONFIRM)
OM_EXPORT(MH_C_DELIVERY_ENVELOPE)
OM_EXPORT(MH_C_EITS                    )
OM_EXPORT(MH_C_EXPANSION_RECORD        )
OM_EXPORT(MH_C_EXTENSIBLE_OBJECT       )
OM_EXPORT(MH_C_EXTENSION               )
OM_EXPORT(MH_C_EXTERNAL_TRACE_ENTRY    )
OM_EXPORT(MH_C_G3_FAX_NBPS             )
OM_EXPORT(MH_C_GENERAL_CONTENT         )
OM_EXPORT(MH_C_INTERNAL_TRACE_ENTRY    )
OM_EXPORT(MH_C_LOCAL_DELIV_CONFIRM     )
OM_EXPORT(MH_C_LOCAL_DELIV_CONFIRMS    )
OM_EXPORT(MH_C_LOCAL_NDR               )
OM_EXPORT(MH_C_LOCAL_PER_RECIP_NDR     )
OM_EXPORT(MH_C_MESSAGE                 )
OM_EXPORT(MH_C_MESSAGE_RD              )
OM_EXPORT(MH_C_MTS_IDENTIFIER          )
OM_EXPORT(MH_C_OR_ADDRESS              )
OM_EXPORT(MH_C_OR_NAME                 )
OM_EXPORT(MH_C_PER_RECIP_DR            )
OM_EXPORT(MH_C_PER_RECIP_NDR           )
OM_EXPORT(MH_C_PER_RECIP_REPORT        )
OM_EXPORT(MH_C_PROBE                   )
OM_EXPORT(MH_C_PROBE_RD                )
OM_EXPORT(MH_C_RD                      )
OM_EXPORT(MH_C_REDIRECTION_RECORD      )
OM_EXPORT(MH_C_REPORT                  )
OM_EXPORT(MH_C_SECURITY_LABEL          )
OM_EXPORT(MH_C_SESSION                 )
OM_EXPORT(MH_C_SUBMISSION_RESULTS      )
OM_EXPORT(MH_C_SUBMITTED_COMMUNIQUE    )
OM_EXPORT(MH_C_SUBMITTED_MESSAGE       )
OM_EXPORT(MH_C_SUBMITTED_MESSAGE_RD    )
OM_EXPORT(MH_C_SUBMITTED_PROBE         )
OM_EXPORT(MH_C_SUBMITTED_PROBE_RD      )
OM_EXPORT(MH_C_TELETEX_NBPS            )

/* Object Identifier (Elements component) */

        /* Content Type */
OM_EXPORT(MH_CTO_INNER_MESSAGE         )
OM_EXPORT(MH_CTO_UNIDENTIFIED          )

        /* External EITs */
OM_EXPORT(MH_EE_G3_FAX                 )
OM_EXPORT(MH_EE_G4_CLASS_1             )
OM_EXPORT(MH_EE_IA5_TEXT               )
OM_EXPORT(MH_EE_MIXED_MODE             )
OM_EXPORT(MH_EE_TELETEX                )
OM_EXPORT(MH_EE_TELEX                  )

OM_EXPORT(MH_EE_UNDEFINED              )
OM_EXPORT(MH_EE_VIDEOTEX               )

        /* Rendition Attributes */
OM_EXPORT(MH_RA_BASIC_RENDITION        )

#include "ex_im_export.h"
```

6.10.2 ex_im_export.h

```
/*
 * File:        ex_im_export.h
 */

/* Interpersonal Messaging (IM) Classes */

OM_EXPORT(IM_C_BILAT_DEF_BD_PRT         )
OM_EXPORT(IM_C_BD_PRT                   )
OM_EXPORT(IM_C_EXTERN_DEF_BD_PRT        )
OM_EXPORT(IM_C_G3_FAX_BD_PRT            )
OM_EXPORT(IM_C_G4_CLASS_1_BD_PRT        )
OM_EXPORT(IM_C_IA5_TEXT_BD_PRT          )
OM_EXPORT(IM_C_INTERPERSONAL_MSG        )
OM_EXPORT(IM_C_INTERPERSONAL_NOTIF      )
OM_EXPORT(IM_C_IPM_IDENTIFIER           )
OM_EXPORT(IM_C_ISO_6937_TEXT_BD_PRT     )
OM_EXPORT(IM_C_MESSAGE_BD_PRT           )
OM_EXPORT(IM_C_MIXED_MODE_BD_PRT        )
OM_EXPORT(IM_C_NATIONAL_DEF_BD_PRT      )
OM_EXPORT(IM_C_NON_RECEIPT_NOTIF        )
OM_EXPORT(IM_C_OR_DESCRIPTOR            )
OM_EXPORT(IM_C_RECEIPT_NOTIF            )
OM_EXPORT(IM_C_RECIPIENT_SPECIFIER      )
OM_EXPORT(IM_C_TELETEX_BD_PRT           )
OM_EXPORT(IM_C_VIDEOTEX_BD_PRT          )
```

6.10.3 ex_im_import.h

```
/*
 * File:        ex_im_import.h
 */

/* Interpersonal Messaging (IM) Classes */

OM_IMPORT(IM_C_BILAT_DEF_BD_PRT        )
OM_IMPORT(IM_C_BD_PRT                  )
OM_IMPORT(IM_C_EXTERN_DEF_BD_PRT       )
OM_IMPORT(IM_C_G3_FAX_BD_PRT           )
OM_IMPORT(IM_C_G4_CLASS_1_BD_PRT       )
OM_IMPORT(IM_C_IA5_TEXT_BD_PRT         )
OM_IMPORT(IM_C_INTERPERSONAL_MSG       )
OM_IMPORT(IM_C_INTERPERSONAL_NOTIF     )
OM_IMPORT(IM_C_IPM_IDENTIFIER          )
OM_IMPORT(IM_C_ISO_6937_TEXT_BD_PRT    )
OM_IMPORT(IM_C_MESSAGE_BD_PRT          )
OM_IMPORT(IM_C_MIXED_MODE_BD_PRT       )
OM_IMPORT(IM_C_NATIONAL_DEF_BD_PRT     )
OM_IMPORT(IM_C_NON_RECEIPT_NOTIF       )
OM_IMPORT(IM_C_OR_DESCRIPTOR           )
OM_IMPORT(IM_C_RECEIPT_NOTIF           )
OM_IMPORT(IM_C_RECIPIENT_SPECIFIER     )
OM_IMPORT(IM_C_TELETEX_BD_PRT          )
OM_IMPORT(IM_C_VIDEOTEX_BD_PRT         )
```

6.10.4 ex_import.h

```
/*
 * File:        ex_import.h
 */
#include <xmhp.h>
#include <ximp.h>
#include <xsmp.h>

OM_IMPORT(MH_C_ALGORITHM)
OM_IMPORT(MH_C_ALGORITHM_AND_RESULT)
OM_IMPORT(MH_C_ASYMMETRIC_TOKEN)
OM_IMPORT(MH_C_BILATERAL_INFO)
OM_IMPORT(MH_C_COMMUNIQUE)
OM_IMPORT(MH_C_CONTENT)
OM_IMPORT(MH_C_DELIV_MESSAGE)
OM_IMPORT(MH_C_DELIV_PER_RECIP_DR)
OM_IMPORT(MH_C_DELIV_PER_RECIP_NDR)
OM_IMPORT(MH_C_DELIV_PER_RECIP_REP)
OM_IMPORT(MH_C_DELIV_REPORT)
/*
 * NOTE: the following two names are not in alphabetical order to retain
 * the value assignment as published in X.400 API Version 2.0
 */
OM_IMPORT(MH_C_DELIV_CONFIRM)
OM_IMPORT(MH_C_DELIV_ENVELOPE)
OM_IMPORT(MH_C_EITS                    )
OM_IMPORT(MH_C_EXPANSION_RECORD        )
OM_IMPORT(MH_C_EXTENSIBLE_OBJECT       )
OM_IMPORT(MH_C_EXTENSION               )
OM_IMPORT(MH_C_EXTERNAL_TRACE_ENTRY    )
OM_IMPORT(MH_C_G3_FAX_NBPS             )
OM_IMPORT(MH_C_GENERAL_CONTENT         )
OM_IMPORT(MH_C_INTERNAL_TRACE_ENTRY    )
OM_IMPORT(MH_C_LOCAL_DELIV_CONFIRM     )
OM_IMPORT(MH_C_LOCAL_DELIV_CONFIRMS    )
OM_IMPORT(MH_C_LOCAL_NDR               )
OM_IMPORT(MH_C_LOCAL_PER_RECIP_NDR     )
OM_IMPORT(MH_C_MESSAGE                 )
OM_IMPORT(MH_C_MESSAGE_RD              )
OM_IMPORT(MH_C_MTS_IDENTIFIER          )
OM_IMPORT(MH_C_OR_ADDRESS              )
OM_IMPORT(MH_C_OR_NAME                 )
OM_IMPORT(MH_C_PER_RECIP_DR            )
OM_IMPORT(MH_C_PER_RECIP_NDR           )
OM_IMPORT(MH_C_PER_RECIP_REPORT        )
OM_IMPORT(MH_C_PROBE                   )
OM_IMPORT(MH_C_PROBE_RD                )
OM_IMPORT(MH_C_RD                      )
OM_IMPORT(MH_C_REDIRECTION_RECORD      )
OM_IMPORT(MH_C_REPORT                  )
OM_IMPORT(MH_C_SECURITY_LABEL          )
OM_IMPORT(MH_C_SESSION                 )
OM_IMPORT(MH_C_SUBMISSION_RESULTS      )
OM_IMPORT(MH_C_SUBMITTED_COMMUNIQUE    )
OM_IMPORT(MH_C_SUBMITTED_MESSAGE       )
OM_IMPORT(MH_C_SUBMITTED_MESSAGE_RD    )
OM_IMPORT(MH_C_SUBMITTED_PROBE         )
OM_IMPORT(MH_C_SUBMITTED_PROBE_RD      )
OM_IMPORT(MH_C_TELETEX_NBPS            )

/* Object Identifier (Elements component) */

        /* Content Type */
OM_IMPORT(MH_CTO_INNER_MESSAGE         )
OM_IMPORT(MH_CTO_UNIDENTIFIED          )

        /* External EITs */
OM_IMPORT(MH_EE_G3_FAX                 )
OM_IMPORT(MH_EE_G4_CLASS_1             )
OM_IMPORT(MH_EE_IA5_TEXT               )
OM_IMPORT(MH_EE_MIXED_MODE             )
OM_IMPORT(MH_EE_TELETEX                )
OM_IMPORT(MH_EE_TELEX                  )
OM_IMPORT(MH_EE_UNDEFINED              )
OM_IMPORT(MH_EE_VIDEOTEX               )

        /* Rendition Attributes */
OM_IMPORT(MH_RA_BASIC_RENDITION        )

#include "ex_im_import.h"
```

6.10.5 ex_makeP2.c

```
/*
 * File:      ex_makeP2.c
 *
 * contains:  make_p2( ), make_ipmsid( )
 *            functions to construct a P2 (Interpersonal Message)
 *            OM object
 *
 * NOTE: The following example is not complete.
 */

#include <xom.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"
#include "ex_import.h"

extern  OM_workspace    x4_workspace;

/*
 * make_p2( )
 *
 *     constructs a p2 content with values; the P2 Content returned
 *     by this function is the Interpersonal Message object.
 *
 *     This example uses the minimally required attributes for
 *     an Interpersonal Message object. These attributes are:
 *           Body    [Object(BodyPart)]
 *           This IPM
 *           Subject
 *     Other attributes are defaulted as specified in the specifications.
 *
 *     The BodyPart subobject in the Body attribute in this example
 *     will be of the IA5 Text Body Part class, with attributes:
 *           Repertoire  (defaulted to ia5)
 *           Text
 * INPUT:
 *     id_string   - unique p2 message identifier string
 *     originator  - originator OR Address in string form
 *     subject     - subject of the message
 *     body        - file that contains data for the body part
 * OUTPUT:
 *     p2_object   - filled Interpersonal Message (private) object.
 * CALLS:
 *     - make_ipmsid( )
 * RESULT:
 *     - success or error
 */
```

```
OM_return_code
make_p2(    char                *id_string,
            char                *originator,
            char                *subject,
            bodypart            *body,
            OM_private_object   *p2_object)
{
    OM_return_code      make_ipmsid(char *, char *, OM_private_object *);
    OM_private_object   p2_private;
    OM_private_object   id_object;
    OM_descriptor       p2[10];   /* enough for envelope information
                                   * of interest */
    OM_descriptor       recip[3];
    OM_return_code      status;
    int                 i = 1;

    /* Create a private P2 object */
    if ((status = om_create(IM_C_INTERPERSONAL_MSG, OM_TRUE,
                x4_workspace, &p2_private)) != OM_SUCCESS)
        return(status);

    MAKECLASS(p2, OMP_O_IM_C_INTERPERSONAL_MSG);

    /* P2 IPMS identifier */
    if ((status = make_ipmsid(originator, id_string, &id_object))
                                                    != OM_SUCCESS)
    {
        om_delete(p2_private);
        return(status);
    }

    put_desc(&p2[i++], IM_THIS_IPM, OM_S_OBJECT, &id_object, 0);

    /* Subject */
    put_desc(&p2[i++], IM_SUBJECT, OM_S_TELETEX_STRING, subject,
                                    strlen(subject));
    ENDOBJ(p2[i]);

    /* Insert those values above first before proceeding.  */
    if ((status = om_put(p2_private, OM_REPLACE_ALL, p2, 0,0,0))
                                    != OM_SUCCESS)
    {
        om_delete(p2_private);
        om_delete(id_object);
        return(status);
    }

    /*
     * Release any private or public objects no longer needed
     * as om_put( ) does not release them.
     */
    om_delete(id_object);
    /* P2 Body Part */
    for (i=0; body[i].bodypart_type != END_BP; i++)
    {
        switch (body[i].bodypart_type)  {
            case IA5TEXT_BP:
                put_ia5(p2_private, body[i].bodypart_file);
                break;
            /*
             * ... insert code for other body part types ...
             */
            default:
                break;
        }
    }
    *p2_object = p2_private;
    return(OM_SUCCESS);
}

/*
 * make_ipmsid( )
 *      allocates and returns a private object containing
 *      the originator and id_string.
 *
 * INPUT:
 *      originator  - OR Address string
 *      id_string   - the identifier string
 * OUTPUT:
 *      id_object   - a private object for a IPM Identifier class
 * RESULT:
 *      - success or error
 */
OM_return_code
make_ipmsid(    char                *originator,
                char                *id_string,
                OM_private_object   *id_object)
{
    OM_return_code        str2obj(char *, OM_descriptor **);
    OM_private_object     id_private;
    OM_descriptor         id_public[4];
    OM_descriptor         *address_desc;
    OM_return_code        status;
    int                   i = 1;

    /* create IPM Identifier object */
    if ((status = om_create(IM_C_IPM_IDENTIFIER, OM_TRUE,
                x4_workspace, &id_private)) != OM_SUCCESS)
        return(status);

    MAKECLASS(id_public, OMP_O_IM_C_IPM_IDENTIFIER);
```

```
        if (originator)
        {
            if ((status = str2obj(originator, &address_desc))
                                    != OM_SUCCESS)
            {
                /* clean up and return */
                free(id_public);
                return(status);
            }
            put_desc( &id_public[i++], IM_USER, OM_S_OBJECT,
                        &address_desc, 0);
        }

        if (id_string)
            put_desc(&id_public[i++], IM_USER_RELATIVE_IDENTIFIER,
                     OM_S_PRINTABLE_STRING, id_string, strlen(id_string));
        else
        {
            free(id_public);
            return(ERR_REQ_FIELD_MISSING);
        }

        ENDOBJ(id_public[i]);

        if ((status = om_put(id_private, OM_REPLACE_ALL, id_public,
                             0, 0, 0)) != OM_SUCCESS)
        {
            om_delete(id_private);
            return(status);
        }

        *id_object = (OM_object) id_private;

        return(OM_SUCCESS);
    }
```

6.10.6 ex_read.c

```
/*
 * File:            ex_read.c
 *
 * contains:        getbody( ), read_file( )
 *
 * NOTE: the following example is not complete.
 */

#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"
#include "ex_import.h"

/*
 * getbody( )
 *        reads the string value (the data) of the bodypart
 *        object and put data value in a file.
 *        It returns the bodypart type (ia5, fax, etc) in
 *        the bodypart_type of the bodypart data structure,
 *        as well as the filename in bodypart_file.
 *
 * INPUT:
 *      bodypart_obj    - the bodypart private object
 *      body            - the bodypart struc to return
 * OUTPUT:
 *      body            - bodypart_type & bodypart_file filled in
 * CALLS:
 *      read_file( )
 * RESULT:
 *      Return error or success
 */

OM_return_code
getbody(OM_private_object   bodypart_obj,
        struct bodypart     *bodypart)
{
    OM_return_code          read_file( );
    char                    *filename;
    OM_return_code          status;
    OM_boolean              instance;

    if ((filename = (char *) malloc(80)) == NULL)
        return (OM_MEMORY_INSUFFICIENT);
    /*
     * Each bodypart object might be of a different class with
     * different attributes, and the attribute having the
     * data string is called differently.
     * Hence, need to check the class for each body part.
     */
        if ((om_instance(bodypart_obj, IM_C_IA5_TEXT_BD_PRT,
                                     &instance) == OM_SUCCESS) &&
                instance == OM_TRUE)
    {
        read_file(bodypart_obj, IM_TEXT, filename);
        bodypart->bodypart_type = IA5TEXT_BP;
        bodypart->bodypart_file = filename;
    }
    else if (1)
           /* etc.... insert code .... */
        ;

    return(OM_SUCCESS);
}

/*
 * read_file( )
 *      reads the string value of an attribute of a bodypart object and
 *      inserts it in a file.
 *
 * INPUT:
 *      subject  - the body part object
 *      type     - the attribute type of the string value in
 *                 bodypart object
 *      filename - pointer to space for the filename.
 * OUTPUT:
 *      filename - now has the name of the file containing the string.
 * RETURN:
 *      Error or Success
 */
OM_return_code
read_file(OM_object        bodypart,
        OM_type            type,
        char               *filename)
{
  char                      buf[BUFSIZE];
  FILE                     *file;
  OM_string_length          string_offset;
  OM_string                 data;
  OM_return_code            status;

        tmpnam(filename);
        if ((file = fopen(filename, "w")) == NULL)
            return(ERR_OPEN_FILE);
        /∗ Start reading at beginning of the value. Note that
         ∗ the offset in the value gets updated by the service ∗
         ∗ to be the next position. The buffer size is indicated in
         ∗ the length of the data.
         ∗/

        data.length = BUFSIZE;
        data.elements = (void ∗) buf;
        string_offset = 0;
        if ((status = om_read(bodypart, type, 0,
                          OM_FALSE, &string_offset, &data))
                                              != OM_SUCCESS)
        {
            fclose(file);
            return(status);
        }
        fwrite((char ∗) data.elements, sizeof(char), data.length, file);

        /∗ Read data for bodypart; om_read( ) returns "string_offset"
         ∗ of zero when end of value is reached.
         ∗/
        while (string_offset != 0)
        {
            if ((status = om_read(bodypart, IM_TEXT, 1, OM_FALSE,
                             &string_offset, &data)) != OM_SUCCESS)
            {
                fclose(file);
                return(status);
            }
            fwrite((char ∗) data.elements, sizeof(char),
                                data.length, file);
        }

        fclose(file);
        return(OM_SUCCESS);
}
```

6.10.7 ex_utils.c

```
/*
 * File:        ex_utils.c
 *
 * contains:    str2obj( ), obj2str( ), put_desc( )
 *
 * Note:        This example is not complete.
 */
#include <stdio.h>
#include <xom.h>
#include <xmh.h>
#include <xmhp.h>
#include <ximp.h>
#include "example.h"

/*
 * str2obj( )
 *     converts an OR-Name or OR-Address string with options (P1 and P2)
 *     into a array of descriptors.
 *     This function allocates enough memory for this array which
 *     ends with OM_NO_MORE_TYPES.
 *     The calling program should free this memory when done using
 *     free( ).
 *
 *     For example, the input string could be of this form:
 *
"/C=CA/AD=A_ADMD/PD=A_PRMD/O=A_FIRM/DELIVERY/NONDELIVERY/RECEIPTREQUEST/1"
 *     where those without "=" are options and
 *     the last number is ordinal position on the list of recipients.
 *
 *     The output is a descriptor array whose
 *     type, syntax and value components filled with values.
 *     This array, or public object, can be used as input argument
 *     to om_put( ).
 * INPUT:
 *     string      - the OR Name string with P1 and P2 options.
 * OUTPUT:
 *     namelist    - an array of descriptors.
 * RESULT:
 *     success or error (memory_insufficient)
 */
```

```
OM_return_code
str2obj(char *string, OM_descriptor **namelist)
{
   /*
    * WARNING: This function does not contain the complete
    * and correct code to perform the indicated functionality;
    * in fact, it simply returns
    * an array of descriptors for a single FIXED name string.
    */

    void put_desc( );
    OM_return_code  status;
    OM_descriptor   *list;
    int i = 0;

    if ((list = (OM_descriptor *) calloc(10, sizeof(OM_descriptor)))
                                                 == NULL)
        return(OM_MEMORY_INSUFFICIENT);

    put_desc( &list[i++], MH_T_COUNTRY_NAME,  OM_S_PRINTABLE_STRING,
              "ca", 2);

    put_desc( &list[i++], MH_T_ADMD_NAME,  OM_S_PRINTABLE_STRING,
              "telecom.canada", 14);

    put_desc( &list[i++], MH_T_PRMD_NAME,  OM_S_PRINTABLE_STRING,
              "osiware", 7);

    put_desc( &list[i++], MH_T_ORGANIZATION_NAME,  OM_S_PRINTABLE_STRING,
              "qa", 2);

    put_desc( &list[i++], MH_T_SURNAME,  OM_S_PRINTABLE_STRING,
              "smith", 5);

    ENDOBJ(list[i]);

    *namelist = list;

    return(OM_SUCCESS);
}
```

```
/*
 * obj2str(OM_descriptor *namelist, char *string)
 *
 * DESCRIPTION:
 *     Performs the reverse of str2obj( ) and concatenates the
 *     appropriate attributes in namelist to the string.
 *
 * INPUT:
 *     namelist  - descriptor array with ORAddress or P1/P2 options
 *     string    - space for string is already allocated
 *
 * OUTPUT:
 *     string    - string now has additional values appended
 *
 * RESULT:
 *     success or failure
 */
OM_return_code
obj2str(OM_descriptor *namelist, char *string)
{
    int         i;
    char        *token;
    char        *str;
    char        buf[128];
    OM_value    value;

    str = string = ' ';
    for (i=0; namelist[i].type != OM_NO_MORE_TYPES; i++)
    {
        /* Note: only handling String attributes, and
         * not subobject attributes in a directory distinguished name. */
        if ((namelist[i].syntax && OM_S_SYNTAX) == OM_S_OBJECT)
                continue;

        value = namelist[i].value;

        switch (namelist[i].type) {
             case MH_T_ADMD_NAME:
                token = "A";
                break;
             case MH_T_COUNTRY_NAME:
                token = "C";
                break;
             case MH_T_DOMAIN_TYPE_1:
                token = "DDATYPE1";
                break;
             case MH_T_DOMAIN_VALUE_1:
                token = "DDAVAL1";
                break;
             case MH_T_DOMAIN_TYPE_2:
                token = "DDATYPE2";
                break;
             case MH_T_DOMAIN_VALUE_2:
                    token = "DDAVAL2";
                    break;
                    /* etc */
                 case MH_T_GIVEN_NAME:
                    token = "G";
                    break;
                 case MH_T_INITIALS:
                    token = "I";
                    break;
                 case MH_T_ORGANIZATION_NAME:
                    token = "O";
                    break;
                 case MH_T_ORGANIZATIONAL_UNIT_NAME_1:
                    token = "OU1";
                    break;
                    /* etc for OU2, OU3, OU4 */
                 case MH_T_PRMD_NAME:
                    token = "P";
                    break;
                 case MH_T_SURNAME:
                    token = "S";
                    break;
                 default:
                    token = "?";
                    break;
           }

           strncpy(buf, value.string.elements, value.string.length);
           buf[value.string.length] = ' ';
           str = str + sprintf(str, "/%s=%s", token, buf);
       }

       return(OM_SUCCESS);
   }
```

```
/*
 * put_desc(OM_descriptor *desc_ptr,  OM_type type,
 *                OM_syntax syntax,  void *value,  int len)
 *
 * DESCRIPTION:
 *     Assigns type, syntax input arguments to a descriptor.
 *     Using syntax, this function casts and assigns appropriately the
 *     value argument.
 *
 *     If the value is a string, the value is a (char *), not (OM_string *).
 *
 * INPUT:
 *     desc_ptr - pointer to the descriptor
 *     type     - the attribute type
 *     syntax   - the attribute syntax
 *     value    - pointer to the value
 *     len      - length of a string, null otherwise
 *
 * OUTPUT:
 *     desc_ptr - pointer to the descriptor with its components assigned
 *
 * RESULT:
 *     (none)
 */
void
put_desc(OM_descriptor *desc_ptr,
            OM_type type,
            OM_syntax syntax,
            void *value,
            int len)
{
    desc_ptr->type = type;
    desc_ptr->syntax = syntax;
    switch(syntax) {
        case OM_S_BOOLEAN:
            desc_ptr->value.boolean =  (* (OM_boolean *) value);
            break;
        case OM_S_ENUMERATION:
            desc_ptr->value.enumeration = (* (OM_enumeration *) value);
            break;
        case OM_S_INTEGER:
            desc_ptr->value.integer = *(OM_integer *)value;
            break;
        case OM_S_OBJECT:
            desc_ptr->value.object.object = * (OM_object *) value;
            break;
        case OM_S_BIT_STRING:
        case OM_S_ENCODING:
        case OM_S_GENERAL_STRING:
        case OM_S_GENERALISED_TIME_STRING:
        case OM_S_GRAPHIC_STRING:
        case OM_S_IA5_STRING:
        case OM_S_NUMERIC_STRING:
        case OM_S_OBJECT_DESCRIPTOR_STRING:
        case OM_S_OBJECT_IDENTIFIER_STRING:
        case OM_S_OCTET_STRING:
        case OM_S_PRINTABLE_STRING:
        case OM_S_TELETEX_STRING:
        case OM_S_UTC_TIME_STRING:
        case OM_S_VIDEOTEX_STRING:
        case OM_S_VISIBLE_STRING:
            desc_ptr->value.string.length = (OM_element_position)len;
            desc_ptr->value.string.elements = (void *)value;
            break;
    }
}
```

6.10.8 ex_write.c

```
/*
 * File:        ex_write.c
 *
 * contains:    put_ia5( ), write_file( )
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <xom.h>
#include <xmhp.h>
#include <ximp.h>
#include "ex_import.h"
#include "example.h"

#define BUFSIZE 1000

#ifndef O_RDONLY
#define O_RDONLY    0
#endif

extern OM_workspace x4_workspace;

/*
 * put_ia5( )
 *    Inserts IA5 text of a file a P2 object as a IA5 Body Part object.
 * INPUT:
 *    p2              - the target P2 object
 *    filename        - name of file containing the IA5 text
 * OUTPUT:
 *    p2 now contains an IA5 Body Part object
 * CALLS:
 *    write_file( )
 * RESULT:
 *    Success or Error
 */
OM_return_code
put_ia5(OM_private_object p2,
        char *filename)
{

    OM_return_code write_file(OM_private_object, OM_type,
                                     OM_value_position,
                                     OM_syntax , char *);
    OM_private_object   ia5_object;
    OM_descriptor       ia5desc[2];
    OM_return_code      status;
   /* Construct an IA5 body part object */
    if (( status = om_create(IM_C_IA5_TEXT_BD_PRT, OM_TRUE,
                         x4_workspace, &ia5_object)) != OM_SUCCESS)
        return(status);

   /*
    * Write the whole IA5 text file into the text attribute of
    * ia5_object.
    * Note that there is only one value in this attribute.
    */
    if ((status = write_file(ia5_object, IM_TEXT,
                           (OM_value_position) 1,
                           OM_S_IA5_STRING, filename)) != OM_SUCCESS)
        {   /* clean up and return */
            om_delete(ia5_object);
            return(status);
        }

   /* Put the IA5 body part object into the p2 object.
    * Class attribute not needed since it will be ignored.
    */
    ia5desc[0].type   = IM_BODY;
    ia5desc[0].syntax = OM_S_OBJECT;
    ia5desc[0].value.object.object = ia5_object;
    ia5desc[1].type   = OM_NO_MORE_TYPES;

   /* There can be more than one bodypart, so insert at end
    */
    if ((status = om_put(p2, OM_INSERT_AT_END, ia5desc,
                             (OM_type_list) 0, (OM_value_position) 0,
                             (OM_value_position) 0))
                                            != OM_SUCCESS)
    {
        /* clean up and return */
        om_delete(ia5_object);
        return(status);
    }

   /* Release the storage for the private ia5_object since it is
    * still outstanding after om_put( ) and the handle is still valid.
    */
    om_delete(ia5_object);
    return(OM_SUCCESS);
}
```

```
/* write_file( )
 *  Writes a file into a string value of an attribute of an object.
 *  This function is generic and can be used when
 *  wishing to write data into P2 body parts.
 * INPUT:
 *  subject - the object
 *  type - the attribute type in object into which string is written
 *  value_position
 *       - the value position in the attribute of this new string value.
 *  syntax  - the syntax of this attribute
 *  filename - the name of the file that contains the string value
 * OUTPUT:
 *  (none)
 * RESULT:
 *  Success or Error
 */
OM_return_code
write_file(  OM_private_object     subject,
             OM_type               type,
             OM_value_position     value_position,
             OM_syntax             syntax,
             char                  *filename)
{
        char                 buf[BUFSIZE];
        FILE                 *file;
        OM_string_length     string_offset;
        OM_string            data;
        OM_return_code       status;
        int                  size;

        if ((file = fopen(filename, "r")) == NULL)
            return(ERR_OPEN_FILE);

       /* Start writing at beginning of the value.
        * Note that the element position in the value gets updated
        * by the service to be the next position
        */
        string_offset = 0;
        while ((fgets(buf, BUFSIZE, file)) != NULL)
        {
            data.length = (OM_string_length) strlen(buf);
            data.elements = buf;
            if ((status = (om_write(subject, type, value_position,
                                    syntax, &string_offset, data)))
                                            != OM_SUCCESS)
            {
                fclose(file);
                return(status);
            }
        }
        fclose(file);
        return(OM_SUCCESS);
}
```

6.10.9 ex_example.h

```
/*
 * File:        example.h
 *
 * contains:    MAKECLASS, ENDOBJ
 *              some data structure definitions
 *              CLIENT_NAME
 *              function prototype declarations for the examples
 */

/*
 * Define a data structure to indicate body part type and
 * the file from which to obtain data for the body part.
 */

typedef struct bodypart {
    int         bodypart_type;
    char        *bodypart_file;
} bodypart;

/*
 * buffer size when read/write string value
 */

#define BUFSIZE     1000

/*
 * the bodypart type constants for the body data structure.
 */
#define END_BP            0
#define BILATERAL_BP      1
#define EXTERNAL_BP       2
#define G3FAX_BP          3
#define G4FAX_BP          4
#define IA5TEXT_BP        5
/* etc */

/*
 * MAKECLASS
 *        sets up the class object identifier as
 *        the first descriptor of a public object
 */
#define MAKECLASS(list,class_oid)   \
    list##[0].type=OM_CLASS; \
    list##[0].syntax=OM_S_OBJECT_IDENTIFIER_STRING;\
    list##[0].value.string.elements=(void *)class_oid;\
    list##[0].value.string.length=sizeof(class_oid) - 1;
```

```
/*
 * ENDOBJ
 *       sets up the last descriptor that delimits a public object
 */
#define ENDOBJ(attr)    \
    attr##.type=OM_NO_MORE_TYPES; \
    attr##.syntax=OM_S_NO_MORE_SYNTAXES; \
    attr##.value.string.length = 0; \
    attr##.value.string.elements = 0;

/*
 * Other error codes
 */
#define ERR_OPEN_FILE                1000
#define ERR_REQ_FIELD_MISSING        1001
#define ERR_NOT_EXPECTED_OBJECT      1002

/*
 * Client name
 */
#define CLIENT_NAME "con1"

/*
 * Utilities and functions prototypes in the programming examples
 */

int x4_MAopen(char *client);

int x4_MTopen(char *client);

OM_return_code
MAgetmsg( int    *priority,
        char     *msgid,
        char     *originator,
        char     *recipients[],
        char     *subject,
        bodypart *body);

OM_return_code
MTgetmsg( int    *priority,
        char     *msgid,
        char     *originator,
        char     *recipients[],
        char     *subject,
        bodypart *body);
```

```
OM_return_code
MAmake_msg( int  priority,
        char     *msgid,
        char     *originator,
        char     *recipients[],
        char     *subject,
        bodypart *body);

OM_return_code
MTmake_msg( int  priority,
        char     *msgid,
        char     *originator,
        char     *recipients[],
        char     *subject,
        bodypart *body);

OM_return_code
make_msgid( char    *msg_id,
        OM_object   *msgid_object);

OM_return_code
MAput_recip_p1(OM_descriptor  *value,
               OM_object      *p1recipt);

OM_return_code
MTput_recip_p1(OM_descriptor  *value,
               OM_object      *p1recipt);

OM_return_code
str2obj(char *string, OM_descriptor **namelist);

OM_return_code
obj2str(OM_descriptor *namelist, char *string);

void
put_desc(OM_descriptor *desc_ptr,
        OM_type type,
        OM_syntax syntax,
        void *value,
        int len);

OM_return_code
read_file(OM_object bodypart,
        OM_type     type,
        char        *filename);

OM_return_code
getbody(OM_private_object   bodypart_obj,
        struct bodypart     *bodypart);
```

```
OM_return_code
make_p2( char       *id_string,
         char       *originator,
         char       *subject,
         bodypart   *body,
         OM_private_object   *p2_object);

OM_return_code
make_ipmsdid( char  *originator,
              char  *id_string,
              OM_private_object   *id_object);

OM_return_code
put_ia5( OM_private_object   p2,
         char *filename);

OM_return_code
write_file( OM_private_object   subject,
            OM_type             type,
            OM_value_position   value_position,
            OM_syntax           syntax,
            char                *filename);
```

Index