NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

MULTIMEDIA ELECTRONIC MAIL
ON
THE X.400 MESSAGE HANDLING SYSTEM

by

Sylvain Carrier, B. Tech., ing.

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Applied Science
in
Electrical Engineering

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
University of Ottawa

ISBN 0-612-00521-6

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

À Sylvie et Annie-France

Acknowledgments

It is a pleasure to express the author's sincere appreciation to his supervisor, Dr. Nicolas Georganas, for his support and encouragement.

The author is grateful to all the colleagues and the Electrical Engineering Department Support Staff for their help and support. A special note of thanks is due to Grant Henderson for the useful discussions, valuable advice and tremendous help.

The author would like to acknowledge the financial assistance of the Canadian Forces Post Graduate Training Plan, the guidance of Dr. Gérard Nourry of the CRC and the enthusiasm of Kim Cameron of Zoomit Corp.

Last, but by no means least, the author would like to express his sincere thanks to his wife without whose support and care this work would not have been possible.

Abstract

The proliferation of small and large computer networks and workstations has been the supporting platform for the extensive use and growth of electronic mail. As users became accustomed to the usefulness of electronic mail, they also encountered its limitations with regards to multimedia content and interoperability. An approach that provides a multimedia capability to X.400 that has a built-in interoperability with the Internet is suggested. This approach ensures that existing X.400 and Internet electronic mail facilities can be used to transport and exchange multimedia information by using published interoperability documents.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ADMD	Administration Management Domain
AU	Access Unit
BERKOM	BERliner KOMmunikationssystem
CCITT	International Telegraph and Telephone Consultative Committee (now ITU-T)
CIO	Coordination, Implementation and Operation of Multimedia Services (RACE 2060)
E-Mail	Electronic Mail
EIT	Encoded Information Type
FTP	File Transfer Protocol
IP	Internet Protocol
IPM	Interpersonal Messaging
IPMS	Interpersonal Messaging System
ISO	International Standardization Organization
ITU-T(S)	International Telecommunication Union - Telecommunication (Standardization)
LAN	Local Area Network
MAN	Metropolitan Area Network
MHEG	Multimedia and Hypermedia Expert Group (ISO)
MPEG	Motion Picture Expert Group (ISO)
MD	Management Domain

MHS	Message Handling System
MIME	Multipurpose Internet Mail Extensions
MS	Message Store
MTA	Message Transfer Agent
MTS	Message Transfer System
O/R	Originator / Recipient
OSI	Open Systems Interconnection
PDAU	Physical Delivery Access Unit
PRMD	Private Management Domain
RACE	Research and Development in Advanced Communications Technologies in Europe
RFC	Request for Comments
SMTP	Simple Mail Transport Protocol
TCP	Transmission Control Protocol
UA	User Agent

1. INTRODUCTION

In the past decade, the proliferation of fast, inexpensive, networked computer workstations has produced an explosion in the use of electronic mail [17]. E-Mail has become an important part of business support activities and it will likely take more importance as users find new ways to use it. It is already one of the most widely used services on almost every computer network, including the Internet [10].

When office computers were limited to simple DOS ASCII text, it was normal to exchange E-Mail that would contain only ASCII text. Since computers can now create and manipulate multiple media, users will expect to be able to exchange that information by E-Mail. Exchanging media other than text is already becoming a problem with word processing documents containing figures or images that are embedded. Once saved in ASCII for an E-Mail transfer, those embedded figures and images are lost. It seems that we have already found new ways to use the E-Mail by trying to mix multiple media. More and more, users are starting to request or even demand multimedia mail, and they expect such mail to interoperate [11]. Trade magazines are already excited about it: E-Mail is becoming the medium of choice in major organizations [16], multimedia is one of the most promising new technologies [31], it delivers services more effectively and fascinates users [38].

In 1989, Huitema [20] discussed the issues related to multimedia E-Mail. He suggested that multimedia could include:

- text, with "composition effects" like the use of several fonts or the highlighting of important sentences;
- graphics, in most cases two-dimensional;
- images;
- voice parts, mostly used as annotations within a complex message;
- animations, i.e., moving pictures, either silent or coupled with voice and sounds; and
- structured data, e.g., produced by spreadsheet programs.

Later, Armbrüster [5] was more specific by saying "Working with multimedia systems enables the user to create, edit, transmit, receive, store, retrieve, compute, render, and delete two or more multiple types of information such as data, text, vector graphics, pixel-oriented images, video signals (moving pictures), and audio signals (voice and sound)." The latter definition of multimedia is more accurate and represents realistically what computers can process now.

It also indicated what a multimedia E-Mail system would have to support. Probably most, if not all, of the activities mentioned above. Supporting multimedia activities does not come without problems. Some problems are [8][20][40]:

- the lack of multimedia document standards;
- the lack of multimedia workstations;
- the lack of high speed networks;

- the lack of agreement on standard interchange formats;
- the lack of standard facilities for viewing and composing multimedia mail;
- storing and retrieving multiple separated data streams;
- continuous recording and retrieval of data streams at real-time rates; and
- large file size.

Some of those problems are not as acute as they were a few years ago. Multimedia workstations are now more common with the newer and better PCs, Silicon Graphics workstations or even the latest Macintoshes from Apple. As far as high speed networks are concerned, LANs and MANs have evolved significantly to the point that some people even question the usefulness of such capacity. On the other hand, some problems are still present and quite important in the world of E-Mail such as the last five problems of the list above.

However, not all of those five will be addressed in this thesis. Storing and retrieving multiple separate data streams and the continuous recording and retrieval of data streams at real-time rates will not be addressed. These will be circumvented by restricting real-time delivery within a single workstation. E-Mail messages will be delivered to that station in a store-and-forward manner and in real-time only within the station. High speed networks and multimedia workstations will not be considered while the lack of multimedia document standards is partly circumvented by the E-Mail standard selected. The E-Mail

standard provides a standard but inflexible structure that can be considered a basic document standard. Enhancement to the multimedia E-Mail system such as logical synchronization [13] will require the use of a multimedia document standard (such as MHEG [26]). This is a complex issue by itself and will not be addressed.

The lack of agreement on standard interchange formats and the lack of standard facilities for viewing and composing multimedia mail will be addressed while the large file size will be left for future work. Solving those problems is certainly an important part of a successful multimedia E-Mail implementation but it should not be forgotten that the key requirement of E-Mail is its interoperability [33]. An E-Mail system will not be very useful if it reaches only 10 or 20% of the people to whom you need to communicate with.

This is where the selection of X.400 shows its importance. There have been many prototypes or even working systems that provided multimedia E-Mail; Diamond [39], Andrew [9] and NeXT Mail [41] have all provided multimedia E-Mail at various degrees. However, none of those were based on a CCITT/ISO standard (X.400 has an equivalent ISO standard; ISO 10021 - Message Oriented Text Interchange System, MOTIS). Adherence to a standard is very important for an E-Mail system as it will ensure that the system can communicate with as many people as possible as long as they conform to the standard. In the early eighties, it was recognized that a broad introduction of electronic mail could only be achieved by international standardization [32].

More recent multimedia E-Mail projects have considered that aspect; Agora [29], Montage [17], EuroBridge [18], BERKOM [24] and the RACE CIO (Coordination, Implementation and Operation of Multimedia Services) [6] are all based on X.400.

What can one more multimedia E-Mail project provide? Built-in compatibility with the *de facto* standard in North America: the Internet. Choosing an international standard for its interoperability, greater security and operation features would be appropriate outside North America but here, an E-Mail system must be fully interoperable to the Internet. The goal of this research is to find a way to provide multimedia mail on X.400 in a compatible, interoperable and useable fashion. The multimedia feature addition must be compatible with what already exists in X.400 and with the Internet E-Mail service. It must also be useable; that is not create unnecessary changes of procedures for the user. It must be as similar to the way X.400 E-Mail is done before as it will be done after the multimedia service is installed.

A multimedia E-Mail facility was defined recently for the Internet; the Multipurpose Internet Mail Extensions (MIME [10][11] and its RFC [7]). Along with MIME, new specifications were added to complement the existing Internet/X.400 mapping specifications to include the new elements [1][2][3]. By using this *de facto* standard to provide multimedia mail in X.400, a maximum of interoperability will be provided with a minimum of conversion. This is the new aspect that will provide the benefits of X.400 as far as functionality and

international connectivity are concerned with a lossless Internet connectivity. This will ensure that users of either network can communicate by multimedia mail but also that tunneling through one network to link users of the other is easy (without significant conversion) and still effective (lossless, minimum size, etc). It will also provide standard interchange formats and standard viewing tools.

Chapter 2 of this thesis will briefly introduce the X.400 standard while chapter 3 will present the concept of the solution in using the MIME standard within X.400. Chapter 4 presents the design of the multimedia module, chapter 5 the results obtained and chapter 6 concludes the thesis.

There is one publication resulting from the thesis. A paper was presented at the 5th IEEE Communications Society International Workshop on Multimedia Communications in Kyoto, Japan on 18 May 1994 and was published in the proceedings. The paper described the concepts of the solution and its implementation.

2. ELECTRONIC MAIL UNDER X.400

2.1 Architecture

The X.400 Message Handling System is a CCITT-ISO standard based on the OSI concept. Without going into the details of OSI, let's position X.400 in the stack. The seven layers of the OSI stack are well known and Plattner's [32] simplified model is in **Figure 1**. The application layer provides all the end-user

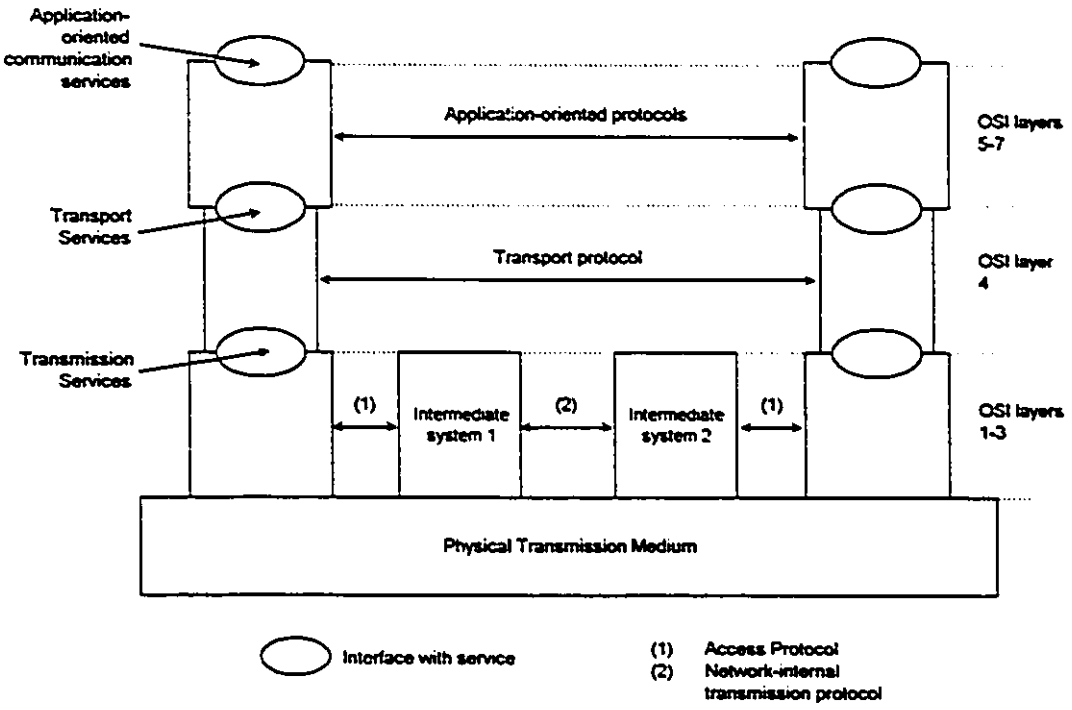

Figure 1 Simplified OSI Reference Model

services typical of distributed computing and this is where we find the X.400 message handling services. Two adjacent X.400 nodes would use an end-to-end protocol in the application layer; thus end-to-end network connection is not required between them (as seen in **Figure 1**). From the perspective of the OSI application layer, all the protocols and functions provided by layers 1 through 6

can be viewed simply as "communication layers", which provide the basic underlying communication function. The lower layers will not be discussed as this is not the area of interest. Not surprisingly, the application layer is typically where most of the complexity in distributed computing resides. In fact, the application layer is often subdivided into a number of sublayers proper to each particular application [33]. As will be seen later, it is the case for X.400.

2.2 The MHS Functional Model

The basic functions of E-Mail such as transport, creation, display and management of messages will be organized into various components. Those components are depicted in the object based functional model in **Figure 2** [23]. In this model, a user is either a person or a computer process. Users are either direct users (i.e. engaged in message handling by direct use of the MHS), or are indirect users (i.e. engaged in message handling through another communication system such as a physical delivery system, that is linked to the MHS). A user is referred to as either an originator (when sending a message) or a recipient (when receiving a message). Message handling elements of service define the set of message types and the capabilities that enable an originator to transfer messages of those types to one or more recipients.

An originator prepares messages with the assistance of his user agent. A user agent (UA) is an application process that interacts with the message transfer system (MTS) or a message store (MS), to submit messages on behalf of a single user. The MTS delivers the messages submitted to it, to one or more

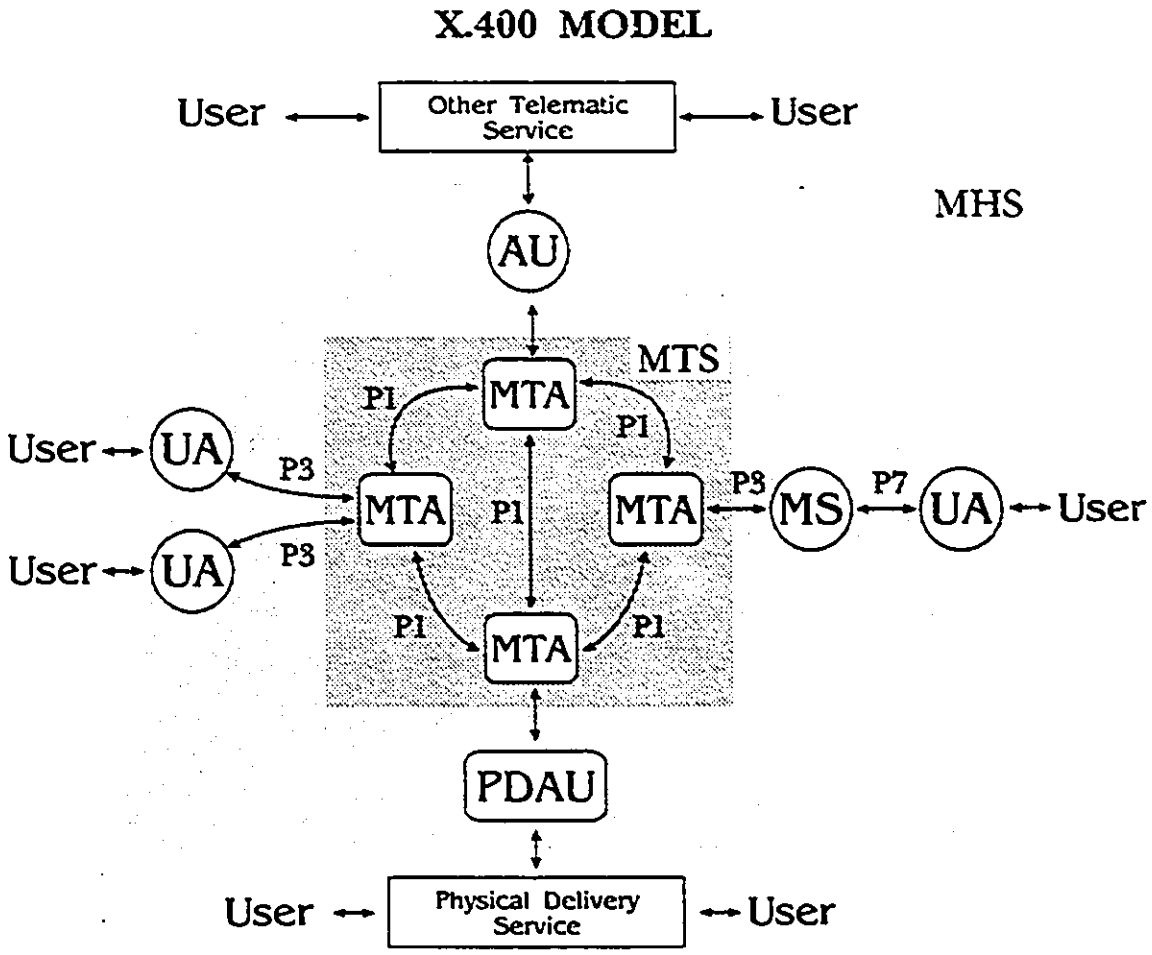

Figure 2 X.400 Functional Model

recipient UAs, access units (AUs), or MSs, and can return notifications to the originator. Functions performed solely by the UA and not standardized as part of the message handling elements of service are called local functions. A UA can accept delivery of messages directly from the MTS, or it can use the capabilities

of an MS to receive delivered messages for subsequent retrieval by the UA. The MS is an optional general purpose capability of the MHS that acts as an intermediary between the UA and the message transfer agent (MTA). The MS is a functional entity whose primary purpose is to store and permit retrieval of delivered messages. The MS also allows for submission from, and alerting to the UA. A MS would prevent a UA from being overflowed with messages upon return to operation after a planned or unplanned down time. The MTA can hold messages for the UA but will deliver them all when the UA becomes available if there is no MS.

The MTS comprises a number of MTAs. Operating together, in a store-and-forward manner, the MTAs transfer messages and deliver them to the intended recipients. It is said to be store-and-forward because messages are not transferred instantaneously from one end system to another. Rather, they are held in the originating end system or one or more intermediate systems until they can be reliably delivered to the intended recipient. The message application is responsible for establishing and managing the underlying communication connection required to transfer messages across a network. The recipient information contained in the envelope of the message is typically used to correctly route messages within the network environment.

That brings up the situation in which the two X.400 nodes are not adjacent (where a relay is required). Then, an intermediate system involves not only the lower layers, but all seven layers since the envelope has to be read at

the application layer (MTA). Figure 3 gives an example of a relay in which the envelope will be read by the X.400 MTA in order to be forwarded [32]. This is the situation that generated application level traffic research efforts [14] as compared to the usual network level relay of Figure 1.

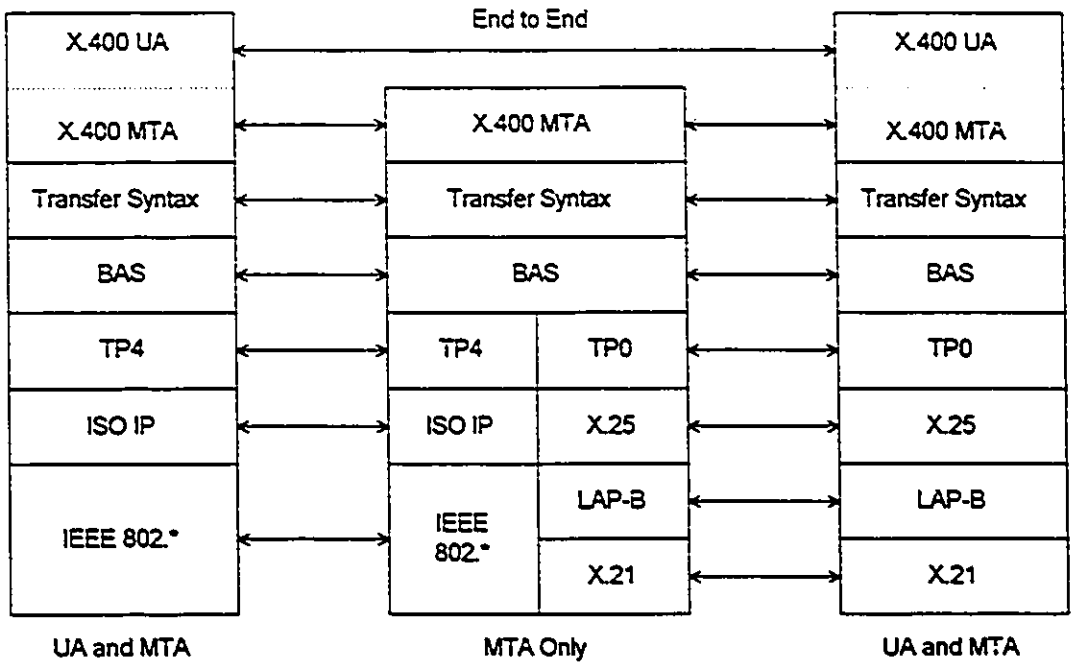

Figure 3 Application Layer Relay

The collection of at least one MTA, zero or more UAs, zero or more MSs, and zero or more AUs operated by an Administration or organization constitutes a management domain (MD, see Figure 4). An MD managed by an Administration (public) is called an administration management domain (ADMD). An MD managed by an organization other than administration is called a private management domain (PRMD). Rules such as: A PRMD is considered to exist entirely within one country and PRMDs are explicitly prohibited from relaying messages between two ADMDs are known to cause

problems to international companies. This is one area where differences are noted between X.400 and MOTIS: MOTIS allows two PRMDs to attach directly to each other across national boundaries.

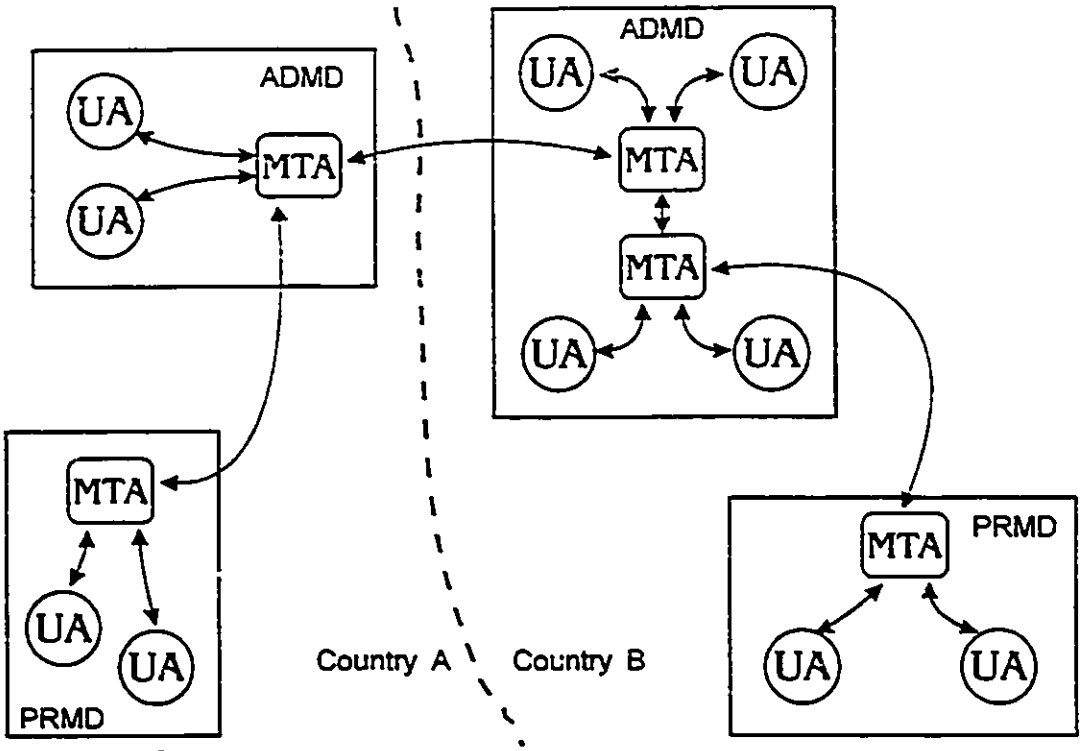

Figure 4 Management Domains

2.3 Structure and Protocols

The envelope-content concept has been introduced but that aspect needs to be discussed further as it will become a key issue of the work performed. A message is made up of an envelope and a content. The envelope carries information that is used by the MTS when transferring the message within the MTS. The content is the piece of information that the originating UA wishes delivered to one or more recipient UAs. The MTS neither modifies or examines the content, except for conversion. Figure 2 showed the protocols that allow the

functional objects to communicate between themselves: The MTS access protocol (P3) used between a remote user-agent and the MTS to provide access to the MTS abstract service, the MS access protocol (P7) used between a remote user-agent and a message store (MS) to provide access to the MS abstract service and the MTS transfer protocol (P1) used between MTAs to provide the distributed

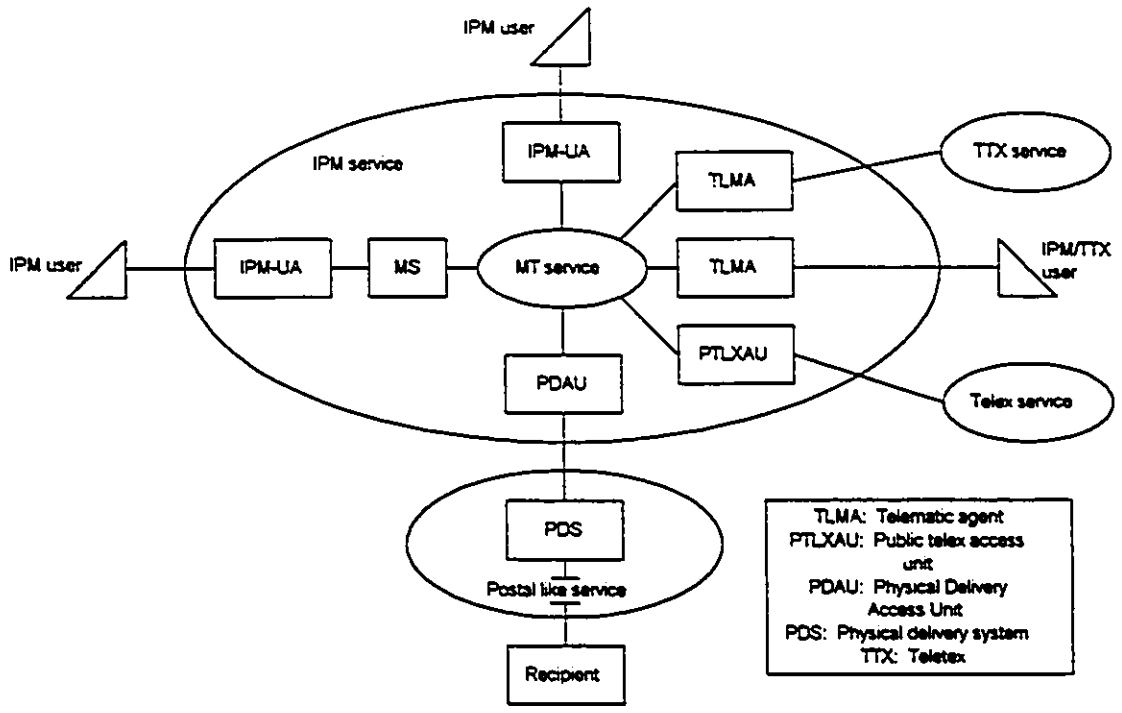

Figure 5 IPM Service Functional Model

operation of the MTS.

There is one more component that has to be described as it is the main application for exchanging mail: Interpersonal Messaging (IPM, X.420 [22]). IPM is a form of message handling tailored for ordinary interpersonal business or private correspondence named P2 (and P22 for version 1988 to version 1984

connectivity). This service, or UA-to-UA protocol, facilitates the transfer of communications between people and thus, in fact, represents E-Mail [32].

The X.400 model can then be tailored for the IPM service as in **Figure 5**. The service is also able to forward communications with components of various types and to convert these communications if necessary (for example, if an

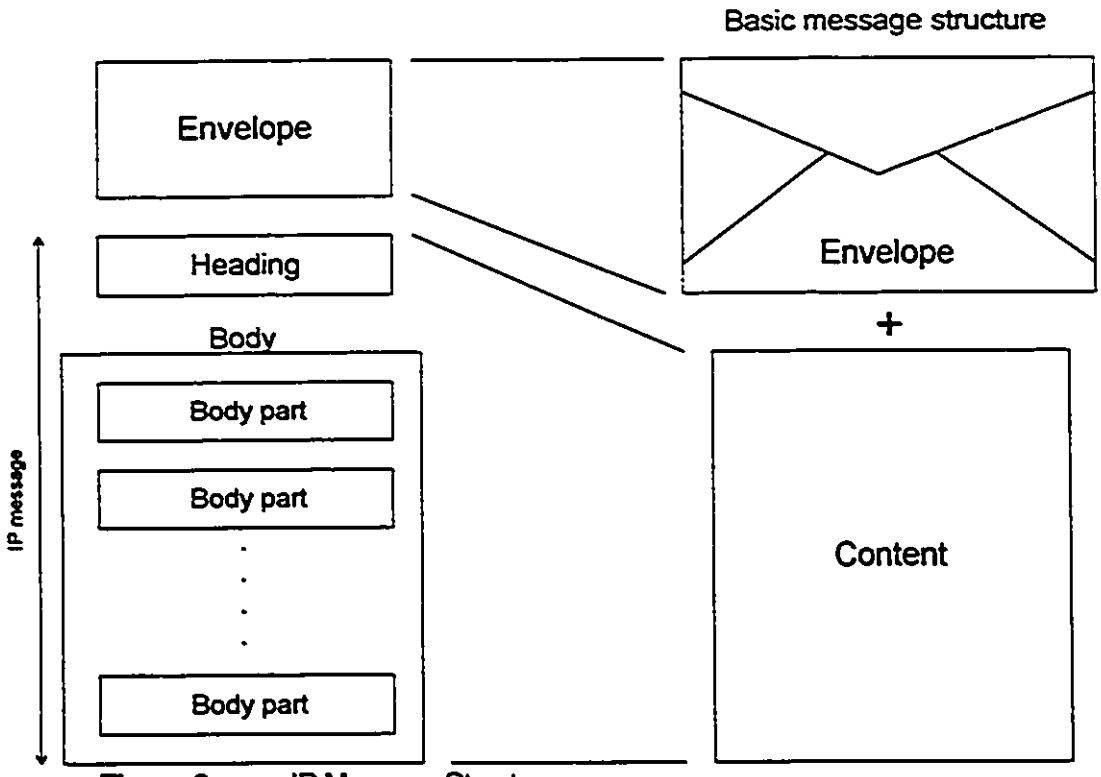

Figure 6 IP Message Structure

output device is unable to restore a communication to its original form). This is done by separating the content of the envelope in a heading and one or more body parts (see **Figure 6**). Such a structure offers an open access to the message and its constituent parts since it is not transmitted as one single proprietary format. Recipients may access all or some of the body parts depending on their respective capabilities.

The heading part could be the "To, From, cc" and subject fields of a memorandum while the text could be in the first body part and a graphic in the second. The body parts can be any of the defined types which are shown in Table I (as of 1992). The types #0 to #6 are self explanatory. Body part #7 represents an information object whose semantics and abstract syntax are nationally defined by a country whose identity is bilaterally agreed by the IPM's originator and all of its potential recipients. This body part is intended for use in domestic communication where the country in question is implicitly that of the originator and all of the potential recipients. The use of this body part is discouraged. It predates the externally defined body part type and is retained for backward compatibility with the 1984 version. The externally defined body part type provides the same capabilities and more, and its use is preferred. An externally defined body part (#15) represents an information object whose semantics and abstract syntax are denoted by an object identifier which the body part carries. It has parameters and data components that represent how the data is encoded (the encoded information type, EIT) and the data itself. The body parts 0 to 6 correspond to basic EITs while the other body parts will have the EIT specified (unless not known, then undefined).

Table I X.400 Body part types

# or Tag	Description
0	ia5-text
3	g3-facsimile
4	g4-class1
5	teletex
6	videotex
7	nationally-defined
8	encrypted
9	message
11	mixed-mode
14	bilaterally-defined
15	externally-defined

On the basis of the externally defined body part type, all body part types are divided into two important classes as follows:

- basic: said of any body part type except Externally Defined. They are all defined by an integer (the tag in Table I), and
- extended: said of the externally defined body part type restricted to any one value of the direct-reference component of the data component of such a body part. They are denoted by an object identifier. In other words, once an externally defined body part is defined with an object identifier, it becomes an extended body part. However, extended and external are often used interchangeably.

An encrypted body part (#8) represents the result of encrypting a body part of a type that is already defined. The parameters of such a body part and the encryption technique that those parameters might identify and parameterize are not defined yet. A message body part (#9) represents an IPM, and optionally, its delivery envelope. It represents a communication which is contained as body part in another communication. It may be used to represent an arbitrary tree structure of forwarded communications. A mixed mode body part (#11) represents a final-form document of the sort that is processable by mixed-mode Teletex terminals and group 4 classes 2 and 3 facsimile terminals. It comprises a sequence of interchange data elements which describe the document's layout structure. A bilaterally defined body part (#14) represents an information object whose semantics and abstract syntax are bilaterally agreed by the IPM's

originator and all of its potential recipients. The use of this body part is also discouraged. It predates the externally defined body part and is retained for backward compatibility with the 1984 version. The externally defined body part type is again preferable.

It can be seen that X.400 lacks body part and encoded information types for multimedia information such as audio, video and images. Text and fax are properly handled but it is far from being fully multimedia. The 1992 version of the recommendation includes a new EBP for audio but it does not impose any encoding but rather leaves room to include its description. That new EBP is more like the File Transfer Body Part discussed in page 48. It does not again fulfill standard multimedia exchange requirements.

2.4 The X.400 Series

The international Message Handling System is usually referred to as X.400 and this is what is being done here. However, it is actually made of several recommendations that cover different aspects of the MHS. The F series recommendations are overviews of the services while the X series are more technical. As of 15 April 1994, the following X and F Series Recommendations were in force:

- X.400 (1993) [Rev.1] [80 pp.] [Publ.: Apr.93] Message handling systems: System and service overview. Note - Same as F.400.
- X.402 (09/92) [Rev.1] [84 pp.] [Publ.: Jul.93] Message handling systems: Overall architecture.

- X.403 (1988) [Blue Book Fasc. VIII.7] [Publ.: Dec.90] Message handling systems: Conformance testing.
- X.407 (1988) [Blue Book Fasc. VIII.7] [Publ.: Dec.90] Message handling systems: Abstract service definition conventions.
- X.408 (1988) [Blue Book Fasc. VIII.7] [Publ.: Dec.90] Message handling systems: Encoded information type conversion rules.
- X.411 (09/92) [Rev.1] [174 pp.] [Publ.: Aug.93] Message handling systems - Message transfer system: Abstract service definition and procedures.
- X.413 (09/92) [Rev.1] [85 pp.] [Publ.: Aug.93] Message handling systems - Message store: Abstract-service definition.
- X.419 (09/92) [Rev.1] [44 pp.] [Publ.: Jul.93] Message handling systems - Protocol specifications.
- X.420 (09/92) [Rev.1] [116 pp.] [Publ.: Oct.93] Message handling systems: Interpersonal messaging system.
- X.435 (03/91) [New] [120 pp.] [Publ.: Sep.91] Message handling systems: Electronic data interchange messaging system.
- X.440 (09/92) [New] [113 pp.] [Publ.: Oct.93] Message handling systems: Voice messaging system.
- X.480 (09/92) [New] [10 pp.] [Publ.: Jun.93] Message handling systems and directory services - Conformance testing.

- X.481 (09/92) [New] [32 pp.] [Publ.: Jul.93] P2 protocol: Protocol implementation conformance statement (PICS) proforma.
- X.482 (09/92) [New] [40 pp.] [Publ.: Jul.93] P1 Protocol - Protocol implementation conformance statement (PICS) proforma.
- X.483 (09/92) [New] [36 pp.] [Publ.: Jul.93] P3 Protocol - Protocol implementation conformance statement (PICS) proforma.
- X.484 (09/92) [New] [42 pp.] [Publ.: Aug.93] P7 protocol - Protocol implementation conformance statement (PICS) proforma.
- X.485 (09/92) [New] [27 pp.] [Publ.: Aug.93] Message handling systems: Voice messaging system protocol implementation conformance statement (PICS) proforma.
- F.400 (08/92) [Rev.1] [80 pp.] [Publ.: Apr.93] Message handling services: Message handling system and service overview. Note - Same as X.400.
- F.401 (08/92) [Rev.1] [19 pp.] [Publ.: Mar.93] Message handling services: Naming and addressing for public message handling services.
- F.410 (08/92) [Rev.1] [9 pp.] [Publ.: Feb.93] Message handling services: The public message transfer service.
- F.415 (1988) [Blue Book Fasc. II.6] [Publ.: Nov.89] Message handling services: Intercommunication with public physical delivery services. Note - Erratum in F.410 (08/92).

- F.420 (08/92) [Rev.1] [14 pp.] [Publ.: Feb.93] Message handling services: The public interpersonal messaging service.
- F.421 (1988) [Blue Book Fasc. II.6] [Publ.: Nov.89] Message handling services: Intercommunication between the IPM service and the telex service. Note - Same as F.85.
- F.422 (1988) [Blue Bock Fasc. II.6] [Publ.: Nov.89] Message handling services: Intercommunication between the IPM service and the teletex service.
- F.423 (08/92) [New] [6 pp.] [Publ.: Feb.93] Message handling services: Intercommunication between the interpersonal messaging service and the telefax service.
- F.435 (03/91) [New] [47 pp.] [Publ.: Jul.91] Message handling: electronic data interchange messaging service.
- F.440 (08/92) [New] [32 pp.] [Publ.: Mar.93] Message handling services: The voice messaging service.

2.5 Sub-Layers, Ports and Service Elements

CCITT's work on 1988 MHS coincided well in timing with the completion of ISO's work on OSI upper-layer architecture. This has led both organizations to adopt a common upper-layer architecture at the end of the 1988 study period. Some of the elements that were in X.400 (1984) were now defined by themselves and no longer part of X.400. That was the case for Remote Operation Service Element (ROSE), Abstract Syntax Notation (ASN.1) and Association Control

Service Element (ACSE). X.400 elements would now be "on top" of standard OSI elements as seen in **Figure 7** [33]. The application independent Application Service Elements (ASEs) defined are ACSE, ROSE and Reliable Transfer Service Element (RTSE). Those general service elements will be used by X.400 but are not specific to X.400 nor are they part of it. The X.400-specific ASEs are:

- Message Administration Service Element (MASE) for administration port services. It is present on MTA and MS objects

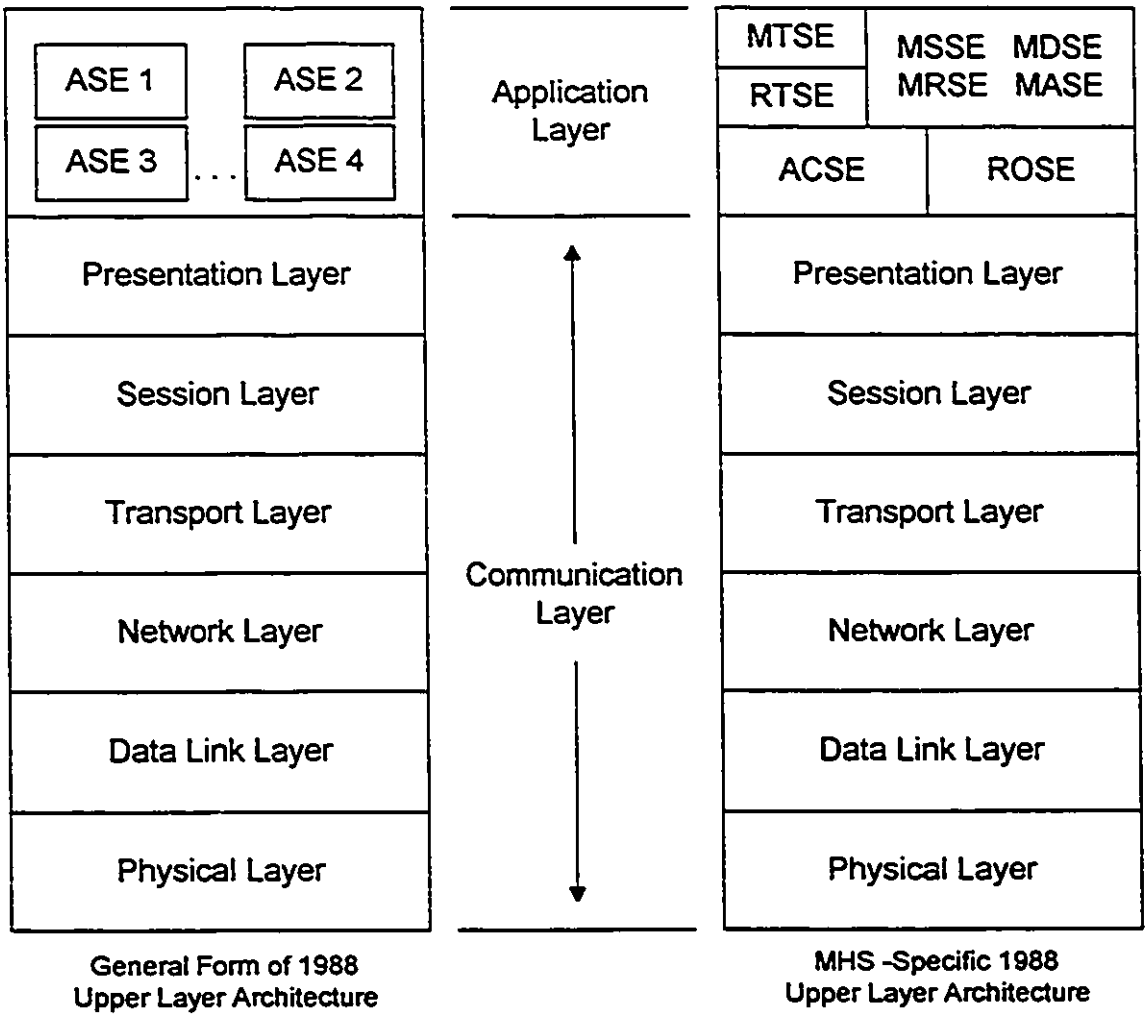

Figure 7 1988 OSI upper-layer architectures

to enable an MTS user to set or change any long term service parameters that the MTS may have defined for it and to enable either the MTS user to change its credentials with the MTS, or the MTS to change its credentials with the MTS user.

- Message Delivery Service Element (MDSE); for delivery port services. It is used to transfer messages and report to or from the MTA. It is also used for delivery controls.
- Message Retrieval Service Element (MRSE) for retrieval port services. It consists of operations designed to summarize, list, fetch, and delete messages held in the MS.
- Message Submission Service Element (MSSE) for submission port services. It will support the operations for submitting messages, probes, or reports to the MTS for transfer within the MHS environment.
- Message Transfer Service Element (MTSE) for transfer port services to establish associations and to transmit messages, probes and reports between MTAs.

It can be seen that ports and service elements are closely related; and sometimes, service ports are even called ASEs [33]. Each component of the MHS is modeled as an object whose overall behavior can be described without reference to its internal structure. The services provided by an object are made available at ports. A type of port represents a particular view of the services

provided by an object. A type of port will correspond to a set of abstract-operations which can occur at the port; those which can be performed by the object and those which can be invoked by the object.

A port can be symmetrical, in which case the set of operations performed by the object may also be invoked by the same object, and vice versa. Otherwise, the port is asymmetrical, in which case the object is said to be the supplier or consumer with respect to the type of port. The terms supplier and consumer are used only to distinguish between the roles of a pair of ports in invoking or performing operations. The assignment of the terms is usually intuitive when one object is providing a service used by another object; the service object (e.g., the MTS) is usually regarded as being the supplier, and the user object (e.g., an MTS-user object) is usually regarded as being the consumer. **Figure 8** shows an example in the case of the MS. The indirect submission port is defined entirely in terms of the submission service port and is used only in conjunction with the

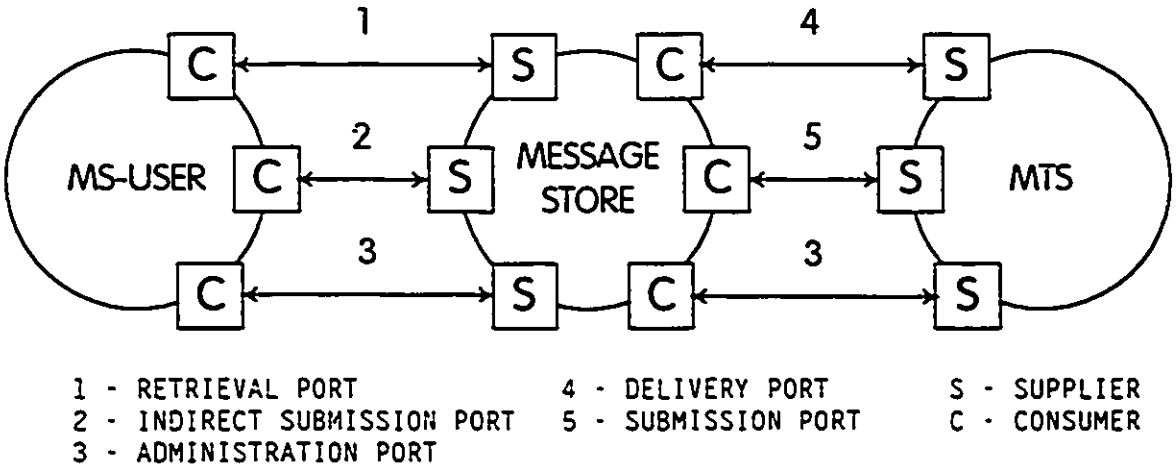

Figure 8 Message store object model with relationships

MS protocol to resubmit to the MTS messages already held in the MS.

Before objects can invoke operations on one another, they must be bound into an abstract association. The binding of an association between the objects establishes a relationship between the objects which lasts until the association is released. An association is always released by the initiator of the association. The binding of an association establishes the credentials of the objects to interact, and the application-context and security-context of the association. The application-context of an association may be one or more types of port paired between two objects.

The model presented is abstract. That is, it is not always possible for an outside observer to identify the boundaries between objects, or to decide on the moment or the means by which operations occur. However, in some cases the abstract model will be realized. For example, a pair of objects which communicate through paired ports may be located in different open systems. In this case, the boundary between the objects is visible, the ports are exposed, and the operations may be supported by instances of OSI communication. Notwithstanding, this will be transparent to a potential user since X.400 is a truly distributed system that meets Tannenbaum's definition of a distributed system [37]:

> *"A distributed system is one that runs on a collection of machines that do not have shared memory, yet looks to its users like a single computer".*

It does not matter where a user is located; the same MTS services are available. The MTS is a single object (made of other objects) with a specific set of services available (see Figure 8); the MS communicates with the MTS and not an MTA. Messages submitted to the MTS are forwarded to the recipient as if that recipient was on the same machine when in fact many nodes and links may separate them.

Now that we have ports and ASEs, those will be grouped together with a description of their interactions to form application contexts (ACs). The AC also determines the role played by a specific ASE amongst the ASEs needed for an application. ACs comprise not only the MHS level ASEs but also those supporting upper layer service elements such as ROSE, RTSE, etc. ACs therefore provide a method for bundling different service functions with different protocol layering requirements. This ability to mix and match protocol components in different ways allows the MHS protocols to achieve both backward compatibility with 1984 MHS, as well as alignment with the 1988 OSI architecture. Table II has a few sample ACs. The (m) indicates that the AC is mandatory while an (o) indicates that the AC is optional. A (c) is for consumer while a (s) indicates that it is a supplier.

Table II Application contexts

	Application context	Constituent ASEs
MTS (P3)	mts-access (m)	MSSE(c), MDSE(c), MASE(c), ROSE, ACSE
	mts-forced-access (m)	MSSE(s), MDSE(s), MASE(s), ROSE, ACSE
	mts-reliable-access (o)	MSSE(c), MDSE(c), MASE(c), ROSE, RTSE, ACSE
	mts-forced-reliable-access (o)	MSSE(s), MDSE(s), MASE(s), ROSE, RTSE, ACSE
MS (P7)	mts-access (m)	MSSE(c), MRSE(c), MASE(c), ROSE, ACSE
	mts-reliable-access (o)	MSSE(c), MRSE(c), MASE(c), ROSE, RTSE, ACSE

2.6 Services

The components of X.400 such as ports, ACs and ASEs, will provide the following services:

for MTS access (P3),

- MTS-bind and MTS-unbind; enables either the MTS-user to establish an association with the MTS, or the MTS to establish an association with the MTS user or to enable the release of an establish association by the initiator of the association. Abstract operations other than MTS-bind can only be invoked in the context of an established association.
- Message-submission, probe-submission, cancel-deferred-delivery and submission control (MSSE, submission port); for submission of messages and probes, request a cancellation of a message previously submitted and constrain the use of the submission port by the MTS-user.

- Message delivery, report-delivery and delivery control (MDSE, delivery port); for delivery and acknowledgment of messages and to constrain the use of the delivery port; and
- Register and change-credentials (MASE, administration port); to change long term parameters and to change credentials in both directions.

for the MS (P7),

- MS-bind and MS-unbind; as for P3.
- Message-submission, probe-submission, cancel-deferred-delivery and submission control (MSSE, submission port); as for P3.
- Summarize, list, fetch, delete, register-MS and alert (MRSE, retrieval port); for database-like operations on the MS and to control automatic actions, defaults, credentials, etc. and to alert the MS-user of a new entry; and
- Register and change-credentials (MASE, administration port); as for P3.

for the MTS protocol (P1),

- MTA-bind and MTA-unbind; as for P3; and

- Message-transfer, probe-transfer and report-transfer (MTSE, transfer port); for the transfer of messages and probes and their acknowledgments. **Figure 9** shows the ports and modules of an MTA.

for the IPMS (P2) using origination, reception and management ports,

- Origination operations invoked by the user and performed by the IPMS; OriginateProbe, OriginateIPM and OriginateRN; for the origination of probes, messages and RN (Receipt Notification).
- Reception operations invoked by the IPMS and performed by the user; ReceiveReport, ReceiveIPM, ReceiveRN and ReceiveNRN for the reception of reports, IPM, RN and NRN (non-receipt notification); and

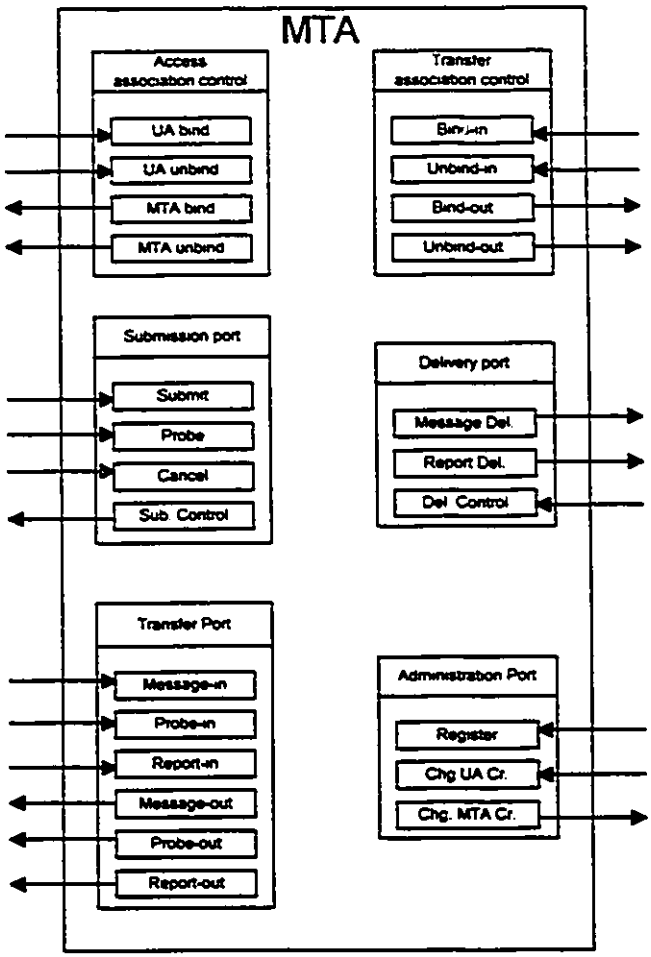

Figure 9 Ports and modules of an MTA

- Management operations invoked by the user and performed by the IPMS; ChangeAutoDiscard, ChangeAutoAcknowledgment and ChangeAutoForwarding for the enabling or disabling of auto-discard, auto-acknowledgment and auto-forwarding.

With the ports and supplier-consumer relationships, our IPMS model will now look as in **Figure 10**.

Services can be divided into two service classes:

- MTS: Basic, submission and delivery, conversion, query and status and inform (see **Table III** for basic services); and

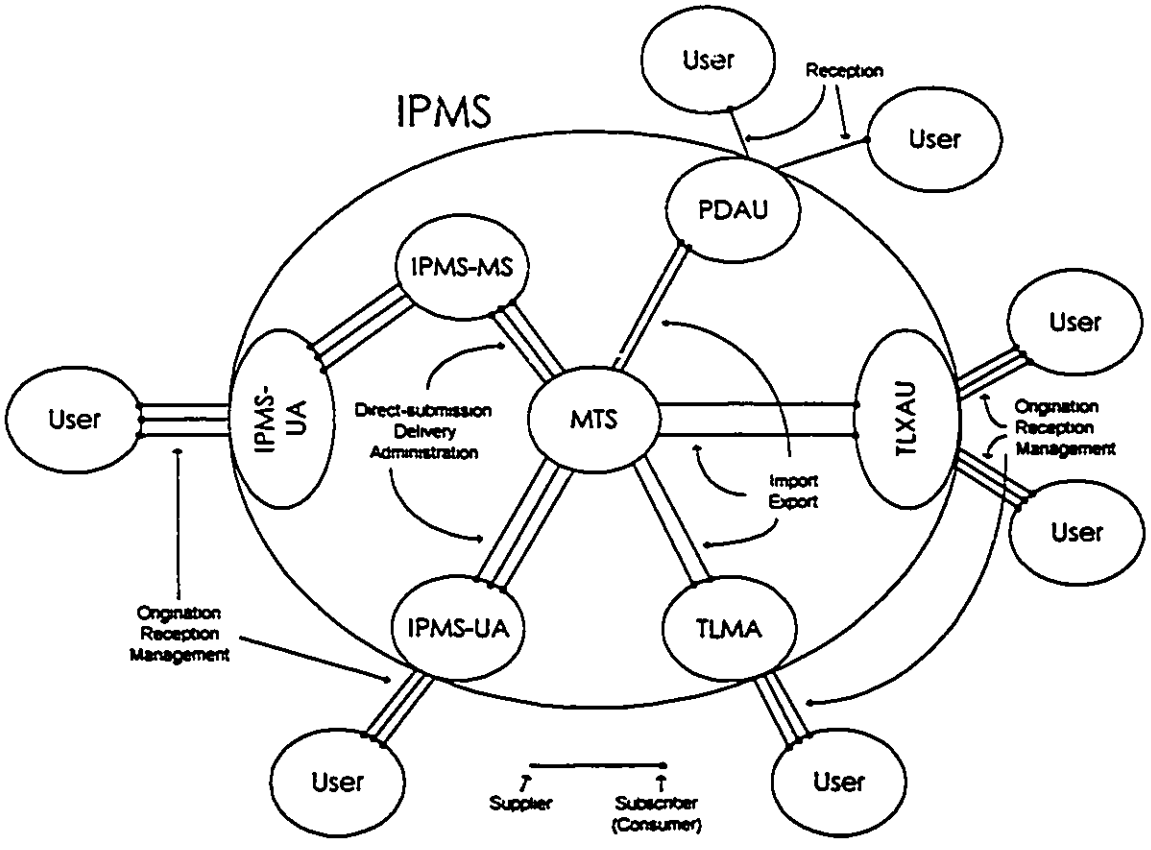

Figure 10 Refined IPMS model

Table III MTS Service elements

Basic Services	
Content-type indication	Specified by originating UA
Converted indication	Specifies any conversions performed on message being delivered
Submission/delivery time stamp	Submission and delivery time are supplied with each message
Message identification	Unique identifier for each message
Nondelivery notification	Message cannot be delivered
Registered encoded informtation type	Allows UA to specify types that can be delivered to it
Original encoded information types	Specified by submitting UA and supplied to receiving UA

- IPM: Basic, submission and delivery and conversion, cooperating IPM UA action, cooperating IPM UA information conveying, query, status and inform (see **Table IV** for basic and essential services).

Stallings [35] has a good overview of X.400 including a list of service elements (not to be confused with elements of service) for the MTS and the IPMS. His services summary is based on the 1984 services but it is still quite informative. It shows the division of services in the three existing categories: *Basic* for services inherent to the MHS and that must be implemented, *Essential*

Table IV IPM Service elements

Basic Services	
IP-message identification	Assign reference identifier to each message content sent or received
Typed body	Allows nature and attributes of message body to be conveyed along with body
Essential Optional Services (for both origination and reception)	
Originator indication	Identifies the user that sent message
Primary and copy recipients indication	Allows UA to specify primary and secondary recipients
Replying IP-message indication	Specifies an earlier message to which this is a reply
Subject indication	Description of message

Optional user facilities must be offered by the service provider but it is up to the user to select or not select the option and *Additional optional user facilities* that may or may not be offered by the provider (tables III and IV are updated extracts from Stallings, details are in the standard [23]).

2.7 Physical Configurations

The possible physical configurations of the MHS, i.e. how the MHS can be realized as a set of interconnected computer systems, are basically unbounded. However, a few configurations are worth discussing since they possess particular characteristics. For example, it might be required to have a co-located AU and MTA if the protocol that governs their interaction is not standardized. On the other hand, no purpose is served by co-locating several MTAs because a single MTA serves multiple users and the purpose of an MTA is to convey objects between, not within a messaging system.

Some configurations would also make the MHS not distributed if a single MTA is used for multiple users on one computer. The fully centralized diagram of Figure 11 is certainly not a distributed system anymore, even if it is still an X.400 MHS. The fully distributed diagram of Figure 11 is a familiar one but many other scenarios are possible. For example, we could have distributed UAs with centralized storage and transfer system. That could be the situation on an LAN using X.400 for the exchange of E-Mail. We could also have distributed UAs and MSs and a central transfer system. That could be the situation where

file servers on different LANS provide storage, workstations provide UAs and the MTA could be located on a minicomputer or mainframe.

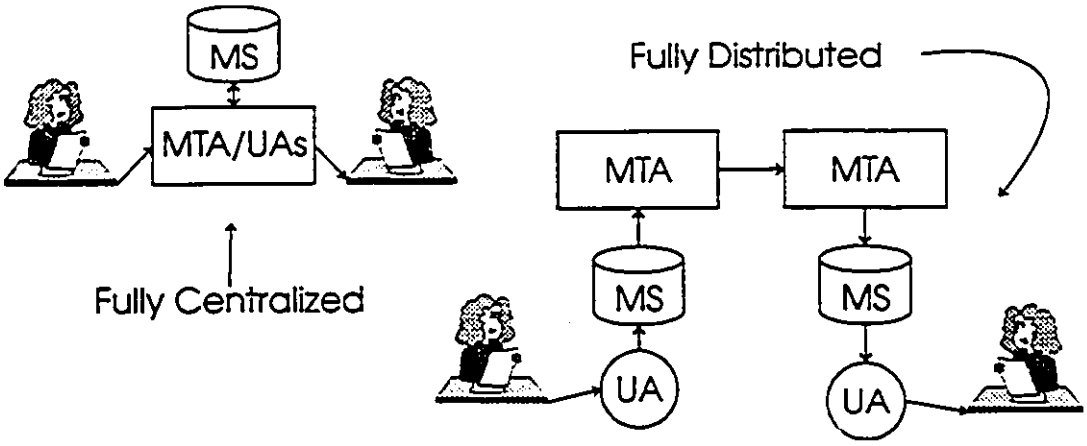

Figure 11 Possible configurations

3. MULTIMEDIA E-MAIL UNDER X.400

3.1 Options

Chapter 2 introduced X.400 and covered the important elements and services of the system. But, carrying multimedia information using the existing elements and services is not possible due to the lack of body parts and encoded-information types. Changing the standard is certainly not desirable: it is a complex standard and it is difficult to reach an international agreement to change it. However, it is not required to add elements or services to carry multimedia information. A means to include multimedia information as part of the standard E-Mail message using standard elements and services is what must be done. This will not require any change to the standard but only for the application at the local level. The concept that follows and its implementation are summarized in the author's paper [15].

A few options should be considered. They are:

a) Use a proprietary format that would include all information and all media and be read by a specific software tool;

b) Use a set of newly defined X.400 body parts that would allow the transport of the information in an open fashion;

c) Use the MIME message format and content-types (body parts) for **encapsulation** in an extended body part (EBP, body part 15); or

d) Use the MIME content-types for **mapping** in separate extended body parts.

3.1.1 Proprietary Format

A multimedia authoring tool (such as a presentation software) could be used to assemble a presentation that would become the multimedia message. Such a tool would generate a proprietary file format that would contain not only all media but also the information on how they are synchronized in time and space (otherwise, a mechanism must be added to provide synchronization).

A tightly coupled presentation software [41] would provide an editing facility for all media and would be very convenient to use. Encoding and decoding of the information and media would be in "native" mode and likely be very efficient. However, such an implementation would not be open: Users would be required to purchase specific software to read such a message. It would not be possible to access only a portion of the information as everything would be enclosed in one proprietary file.

This option would be somewhat easy to implement but it would also defeat the purpose of the standard. The standard is meant to provide message services to users in a heterogeneous environment. With this approach, users would be forced to use a specific software that may not be available on all platforms. By having an encoding scheme that is not published, users are vulnerable. They must depend on the software developer for the access of the MHS on a specific platform or even for the way it is accessed. Exchanging messages with the Internet (MIME) would not be possible and it would not follow the MIME approach (MIME strongly discourages the mixing of multiple

media in a single body). As mentioned by Einar Stefferud [36]: Open markets usually prevail.

3.1.2 New Body Parts

It is tempting to select the best encoding schemes and make them the new MHS body parts. That process would be totally independent of the existing schemes and standards in use. It would be open and it would be possible to retrieve portions of a message but it may have a limited interoperability with the Internet. Also, the selection of the new body parts and the change to the standard would have to be done by international agreements which are difficult or at best long to obtain.

3.1.3 MIME Format

Here we could do one of two things: encapsulate a whole MIME message in a body part or map all MIME content-types in corresponding body parts (see Figure 12).

Both approaches would allow us to capitalize on MIME related research and development being performed around the world (both academic and commercial). However, in order to use commercial X.400 User Agents, they will need to incorporate most of the functionality of a MIME based User Agent. This will result in additional coding and increased size of the executables. However, that will be required regardless of the method used (mapping - encapsulation).

We will assess each approach with the following criteria: effect on gateways (& tunneling: when a message of a given type goes through a system of

another type by being encapsulated within a message of the other type), performance (overhead, redundance), ease of implementation, interoperability, and functionality.

3.1.4 Encapsulating

First, it must be decided if the Internet header is repeated in the encapsulation. That would be a duplication of information but the encapsulated message would not have to be altered in any way for transmission over the Internet.

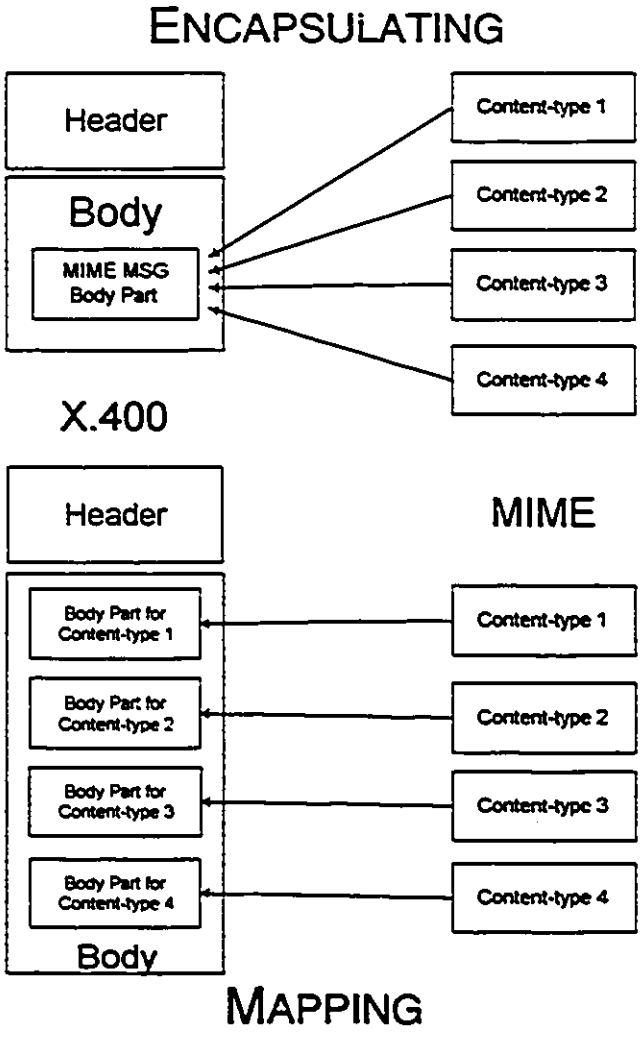

Figure 12 Encapsulating vs Mapping

For gateways and tunneling, encapsulation should require less work than mapping as only one body part has to be "mapped" into one message. However, existing gateways will not be able to do such a task without modification. They can map various parts of a message to other parts but they cannot take one part and "map" it as a new Internet message (the modification might not be

worse than one for mapping though). The amount of work also depends on whether the address was encapsulated or not. If the address is encapsulated, it is less work than having to "map" X.400 header information into the Internet message. However, such a mapping is already defined and can be handled by gateways (Chap 4 of RFC 1327 [19]). The Internet header should not be repeated in the encapsulation as the mapping is already defined and can already be used.

Encapsulating could be faster than mapping as the encapsulated message does not have to be modified part by part. This performance improvement might, however, be negligible. Then, encapsulation would maintain the transfer encoding of the MIME message and that would cause a one third increase in message size (see section 3.6). It would be more expensive, it would take more time and probably cause more congestion than a process that does not keep the transfer encoding. Also a standard X.400 UA could not be used as the form that would be transported by X.400 would not be in standard X.400 format.

Implementation should be easier as MIME authoring tools are becoming more and more available. MIME enabled User Agents are being developed around the world for DOS, MS Windows, MAC, UNIX, XWindows, Amiga, etc. Many of these user agents will be released with source on the Internet. Additionally, Lotus, Microsoft, IBM and numerous other commercial E-Mail vendors are working on adding MIME capabilities to their products. The implementation may then require less effort than with the mapping scheme as MIME enabled software can be acquired and patched in the X.400 environment.

Notwithstanding its benefits, interoperability will not be served well with encapsulation. Although MIME is not proprietary, it is a different "format" that we would be using within X.400. If we were to send a multimedia message to another X.400 user who does not have the multimedia capability, he may have some problems. All the information would be contained in one body part, including the plain text part, and it might be more difficult for a non-multimedia user to read that part. That becomes more of a functionality issue. We could loose some functionality due to the structure that would be used. In this case, it would not be possible to use the defined X.400 body parts as the whole message (with all the parts) would be in one externally defined body part.

If X.400 is ever given an external body part reference mechanism such as the MIME Message/External Body content type, encapsulation would not allow the X.400 network to know that it has to retrieve an external body part. The file pointer would not be read until "decapsulation". This would fall in the functionality-interoperability category but X.400 does not have that capability at this time. This may create a problem when the X.400 user agent "decapsulates" the information and ends up with an Internet external reference content-type but finds itself incapable of retrieving it because of the non-Internet environment!

By encapsulating a MIME encoded message, we benefit from the well defined MIME standard (in particular the content type encoding specification and standardization), we retain the enhanced security benefits offered by X.400

and we should get a much easier implementation. But we may loose some functionality and interoperability.

3.1.5 Mapping

Much work has been done in the area of the Internet (RFC 822) and X.400 mapping. RFC 1327 [19] detailed the mapping between them but did not cover the support for multimedia content. This was covered in RFC 1494 [1], RFC 1495 [2], and RFC 1496 [3]. The multimedia mail system developed for RACE [6], in BERKOM [24] and for Eurobridge [18] are using EBPs that can also be mapped.

It is expected that mapping would require a significant amount of work for gateways (initial development and maintenance for new body parts / content types to follow MIME). All body parts have to be mapped to MIME types and in the case of tunneling, it may have to be done twice. This may affect performance but mapping of a few parts -without canonical conversion- should not be significant. The user will not be concerned with this aspect as E-Mail is not real-time.

Implementation will require the effort of developing a multimedia module as this form of X.400 messaging is not available yet. However, some body parts are already defined in the standard and their use should not be an issue. It should also be possible to use the same viewers that many MIME readers are using.

As far as interoperability is concerned, mapping would be the ideal situation. All contents would appear in the native X.400 structure. Access to any part of the message would be possible by any X.400 user, whether is has multimedia capability or not. Displaying a media could be a problem but receiving it and discriminating a given media from the others would be possible.

On the functionality side, there should be no significant concern. Our X.400 message would follow the usual X.400 message structure and access to any of the body parts would be done as with any other message.

3.1.6 Selected option

The use of the MIME content-types would give us a more open approach than a proprietary format and it would better respect the purpose of X.400. New body parts would also have an open architecture but interoperability problems with the Internet would be created. Not to mention the difficult change to the standard.

The encapsulation of a MIME message would solve the interoperability issue but it would also create some difficulties for an internal X.400 exchange as it would not be really open. For example, it would be very difficult for a non-multimedia user to only extract the facsimile content of an encapsulated MIME message. The extraction of text might not be so difficult if it is at the beginning of the encapsulated MIME message. But if the text is placed after pages of unreadable text representing images or audio, retrieving the relevant text

portion will be very tedious and painful. The increased size of the encapsulated message is also a concern.

For those reasons, the use of the MIME content-types converted to X.400 extended body parts in a mapping scheme is the preferred option to provide functional and interoperable multimedia mail over X.400.

3.2 MIME

The Internet community has encountered the same problem that we are facing in X.400 and it was solved with MIME. There were no content-types for multimedia information, no multi-part structure and a strong requirement for compatibility with the existing network. MIME has defined new content-types (body parts) and sub-types that allow the transportation of multimedia information over the existing Internet network. Achieving multimedia E-Mail over X.400 will be easier and much more interoperable with the Internet (MIME) through the use of the MIME content-types to carry the multimedia information.

The MIME document [7] is designed to provide facilities to include multiple objects in a single message, to represent body text in character sets other than US-ASCII, to represent formatted multi-font text messages, to represent non-textual material such as images, and audio fragments, and generally to facilitate later extensions defining new types of Internet mail for use

by cooperating mail agents. One of the most notable limitations of RFC 821/822[1] based mail systems is the fact that they limit the contents of E-Mail messages to relatively short lines of seven-bit ASCII. This forces users to convert any non-textual data that they may wish to send into seven-bit bytes representable as printable ASCII characters before invoking a local mail UA. RFC 822 did not have the multiple body parts structure that X.420 inherently has. This had to be handled by MIME. The new content-types introduced by MIME are:

- Text; for textual information. The primary subtype, "plain", indicates plain unformatted text. The default Content-type for Internet mail is "text/plain; charset=us-ascii". The other defined character set is ISO-8859.
- Multipart; for the case of multiple part entities, in which one or more different sets of data are combined in a single body. The body part must then contain one or more "body parts", each preceded by an encapsulation boundary, and the last one followed by a closing boundary. Each part will contain a header (to indicate the type and subtype) and a body area (data) like an X.400 extended body part. This multipart type is the type that creates an X.400 like multiple body parts structure. The defined subtypes are mixed, alternative,

---

[1] RFC 821 defined the Simple Mail Transport Protocol (SMTP) while RFC 822 has defined the standard format of textual mail messages on the Internet. The analogy to X.400 would be RFC 821 = X.400 and RFC 822 = the IPMS (X.420).

digest and parallel. Mixed is intended for use when the body parts are independent and need to be bundled in a particular order (serially in the order given for example). Alternative indicates that each part is an "alternative" version of the same information. Digest defines a "digest" in the sense of a summary or synopsis such as a collection of extracts of messages for example. Parallel indicates that the order of body parts is not significant. A common presentation of this type is to display all of the parts simultaneously.

- Message; for the encapsulation of another mail message. This type is similar to the X.400 body part 9, "message". However, it has the following subtypes defined: rfc822, partial and external-body. RFC822 indicates that the body part contains an encapsulated message with the syntax of an RFC822 message. The partial subtype is defined in order to allow large objects to be delivered as several separate pieces of mail and automatically reassembled by the receiving user agent. This mechanism can be used when intermediate transport agents limit the size of individual messages that can be sent. The external-body subtype indicates that the actual body data are not included, but merely referenced. In this case, the parameters describe a mechanism for accessing the external data. This is a body part for which no equivalent exists in

X.400 and that could create problems. Such an external body part would have to be retrieved at the boundary between systems by a gateway. It would be impossible within X.400 to retrieve an external-body part of a message using FTP for example. The benefits of this content type is the avoidance of the message size limit imposed by some gateways and as external bodies are not transported as mail, they need not conform to the 7-bit and line length requirements of Internet, but might in fact be binary files.

- Application; to be used for data which do not fit in any of the other categories, and particularly for data to be processed by mail-based uses of application programs. In general, the subtype of application will often be the name of the application for which the data are intended. Two subtypes are defined: octet-stream and Postscript. The former indicates that a body contains binary data and the latter indicates a Postscript "program".
- Image; to indicate that a body contains an image. Two initial subtypes are "jpeg" and "gif".
- Audio; to indicate that the body contains audio data. The initial subtype of "basic" is specified to meet the requirement of interoperability by providing an absolute minimal lowest common denominator audio format. It represents single channel audio encoded using 8-bit ISDN μ-law at a sample rate of 8 KHz.

- Video; to indicate that the body contains a time-varying-picture image, possibly with color and coordinated sound. The defined subtype is "mpeg" and refers to video coded according to the MPEG standard. In general the MIME document strongly discourages the mixing of multiple media in a single body. However, it is recognized that many so-called "video" formats include a representation for synchronized audio, and this is explicitly permitted for subtypes of "video".
- Experimental; is a private value to be used by consenting mail systems by mutual agreement. Any format without a rigorous and public definition must be named with an "X-" prefix and publicly specified values shall never begin with "X-".

3.3 MIME - X.400 Mapping

In order to use MIME content-types, an extended body part has to be created for each MIME content-type that is intended to be used in X.400. This is where the series of RFCs for MIME-X.400 interconnection become important (RFC 1494 [1], 1495 [2] and 1496 [3]). Those RFCs were introduced to allow a lossless interconnection between the newly defined MIME and existing X.400 systems. The lossless aspect becomes even more important when the tunneling procedure is taken into consideration [see section 3.1.3]. The "tunnel" system may not have to display or interpret the data as long as it is capable of rendering the data, "at the other end of the tunnel", unaltered.

The mappings done have been specifically designed to provide optimal behavior for three different scenarios:

- Allow a MIME user and an X.400 user to exchange an arbitrary binary content;
- Allow MIME content-types to "tunnel" through an X.400 relay that is, two MIME users can exchange content-types without loss through an X.400 relay; and
- Allow X.400 body parts to "tunnel" through a MIME relay that is, two X.400 users can exchange body parts without loss through a MIME relay.

In our case, the use of the MIME EBPs would add the following to the list:

- Allow X.400 users to exchange multimedia information.

RFC 1496 addresses the rules for downgrading messages from X.400/88 to X.400/84 when MIME content-types are present in the messages. The other two RFCs of that group will be more useful as they address the equivalencies between 1988 X.400 and RFC-822 message bodies (RFC 1494) and the mapping between X.400 and

Table V MIME to X.400 Table

MIME Content-Type	X.400 Body Part
text/plain - ASCII	ia5-text
text/plain - ISO-8859	EBP - General Text
text/enriched	nil
application/octet-stream	bilaterally-defined
application/post-script	EBP - mime-postscript-body
image/g3fax	g3-facsimile
image/jpeg	EBP - mime-jpeg-body
image/gif	EBP - mime-gif-body
audio/basic	nil
video/mpeg	nil

EBP - Extended Body Part

RFC-822 message bodies (RFC 1495).

Table V is the conversion table for known X.400 and MIME types. The RFC actually includes application/ODA and text/richtext but the former was dropped in the latest MIME RFC and the latter changed to text/enriched. It can also be noted that audio and video are not mapped to X.400. The decision was made at the time not to map audio as there was ongoing work in X.400 that would provide a better audio format [4]. Video was not mapped due to the lack of directions on how to transport it under X.400.

Table VI shows the inverse mapping; from X.400 to MIME. The actual RFC did include voice and ODA but since those were dropped in MIME or X.400,

Table VI X.400 to MIME Table

X.400 Body Part		MIME Content-Type
Basic body parts	ia5-text (BP 0)	text/plain; charset=us-ascii
	g3-facsimile (BP 3)	image/g3fax
	g4-class1 (BP 4)	no mapping defined
	teletex (BP 5)	no mapping defined
	videotex (BP 6)	no mapping defined
	encrypted (BP 8)	no mapping defined
	bilaterally-defined (BP 14)	application/octet stream
	nationally-defined (BP 7)	no mapping defined
	externally-defined (BP 15)	see EBPs below
Extended body parts (EBPs)	General text	text/plain; charset=iso-8859-x
	mime-postsript-body	application/postscript
	mime-jpeg-body	image/jpeg
	mime-gif-body	image/gif

they are not included. However, missing from Table VI, are the message body part (BP 9) and the mixed mode body part (BP 11). The mixed mode body part is related to mixed mode teletex and is not a concern. The message body part is used to encapsulate a message (forwarded IPM); in that case the mapping is applied recursively. If there is no registered mapping for a given X.400 body part, it is mapped to a MIME application/x400-bp. That content-type will contain the raw ASN.1 IPM.Body octet stream that will have to be subjected to the MIME transfer encoding (see section 3.6). This is not desirable as a MIME recipient could not handle such a body part. To truly have a multimedia E-Mail system, it will be required to map the audio and video content-types to X.400 EBPs as in Figure 13. The two shaded boxes are the only basic body part types that will be used. All the others are extended body part types (EBPs). That will provide the openness, the interoperability with MIME but also a truly multimedia system within X.400.

The 1992 X.400 recommendations introduced a new extended body part that has been designed specifically to meet the users' requirements for file transfer. A file-transfer-body-part represents an information object used to convey the contents, and optionally, the attributes of a stored file. The file-transfer-body-part is based on the file model defined in FTAM. Since that new body part is actually an instance of the externally defined body part, it is possible to benefit from its definition in 1988 implementations. The advantage of using file transfer body part type is that, in a unified way, it accommodates

inclusion of additional file related information within the extended body part. It is a standard EBP with extra information.

This EBP is not to be confused with the MIME message/external-body which contains only a reference to the body. The file transfer body part contains

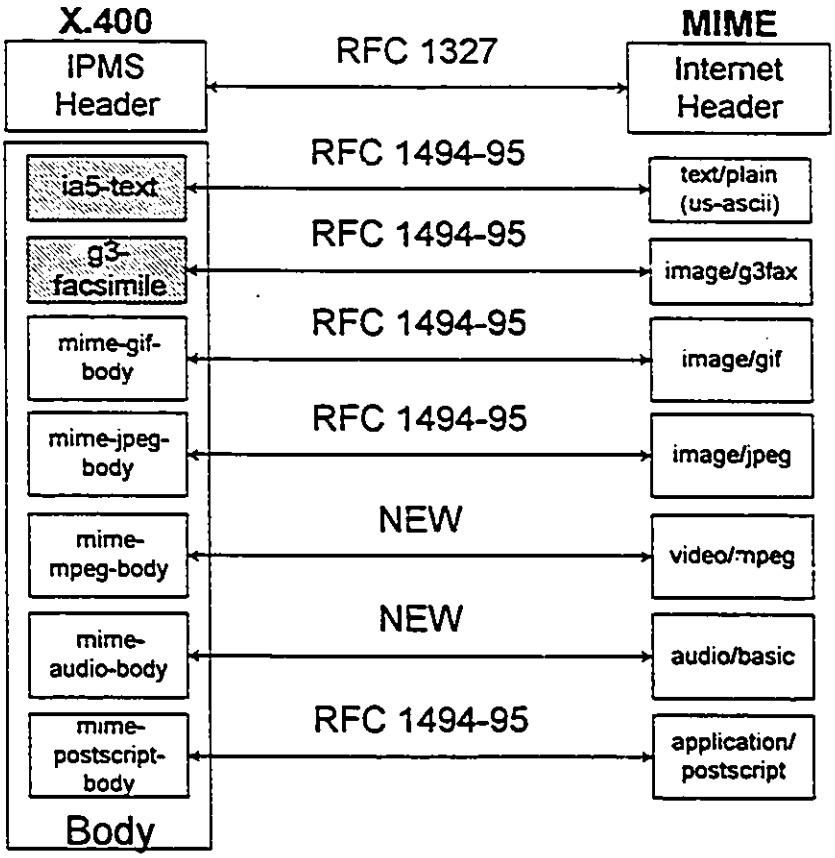

Figure 13 Desired mapping

the body (data). The Electronic Mail Association (EMA) PRMD Operators Committee (POC) has a working group (Message Attachment Working Group, MAWG) that is conducting a study to develop procedures that will allow attachments, such as spreadsheets or word processing documents, to be automatically identified and recognized when delivered between electronic

messaging systems using the file transfer body part. Their work is similar to the work presented here in that it uses an EBP to transport and identify the data. This new body part would not be required nor useful for our application. We do not need to provide more information than that it is a MIME equivalent body part as the viewer is selected at the recipient's station according to the recipient's preferences. This selection process will be covered in the design section.

3.3.1 Object Identifiers (OID) Registration

In order to realize the benefits of interoperability, many infrastructural elements need to be in place. These include global identification and recognition of application data. In order for a data identification method to be useful on an international messaging level, there must be recognized and published, globally unique object identifiers for data types. Currently, there is no widely adopted method for vendors to register and publicize their application data types and identifiers. EMA's eventual goal is to provide application, document and messaging vendors with consensus user input and guidance regarding how to implement and support object identification and encoding. MIME defines a registration process which uses the Internet Assigned Numbers Authority (IANA) as a central registry for such values. Once an entity has registered an OID to describe an X.400 body part, it should ensure having a corresponding entry with the IANA. This issue is not very important for a prototype but certainly is when widespread implementation is planned. Its impact should not

be that significant since the X.400 equivalent body parts supported will follow only those that are standard in MIME.

3.4 Internetworking

Now let's examine how the selected process will work with existing systems. Figure 14 is a graphical representation of the internetworking process

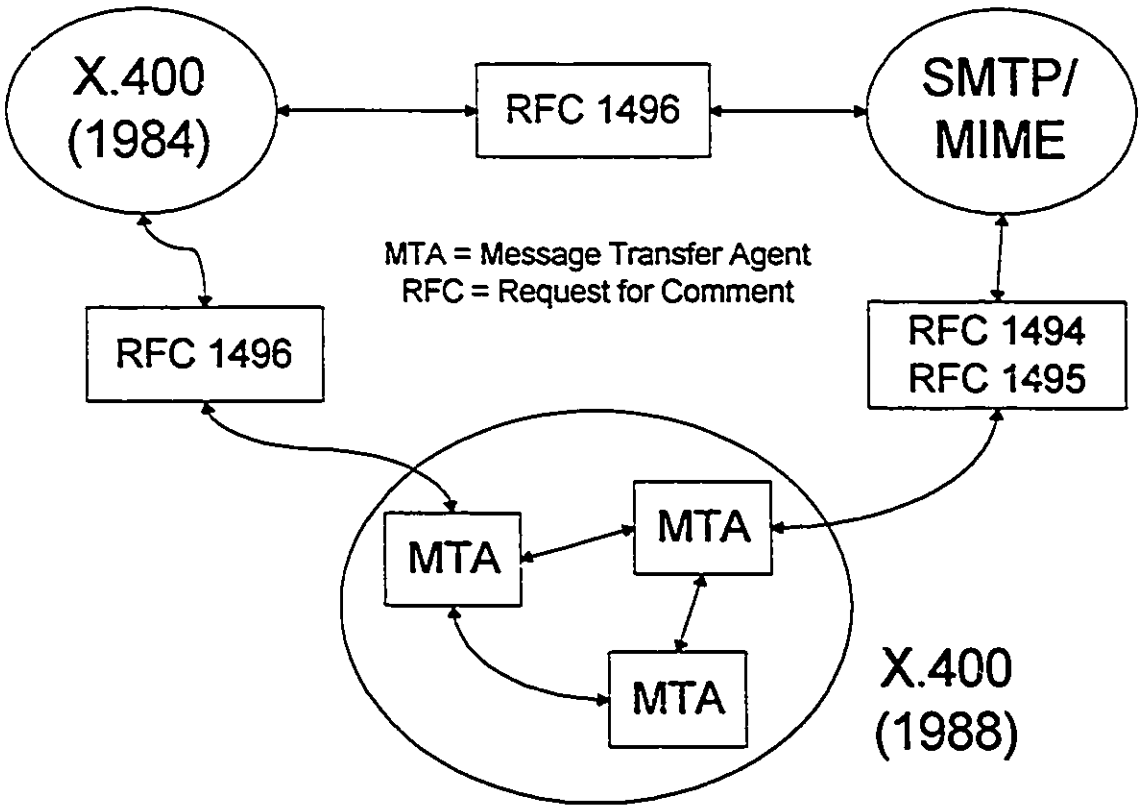

Figure 14 Internetworking

that will be in place. Within X.400 (1988) the process is entirely following the X.400 standard and using EBPs that carry a MIME equivalent multimedia body part. If an exchange (or tunnel) is required with SMTP/MIME, RFC 1494 and 1495 will be followed. RFC 1495 [2] is complementary to RFC 1327 [19] as it focuses on translation of the message body. RFC 1494 [1] describes the content of the "IANA MHS/MIME Equivalence table" found in RFC 1495 and defines the

initial configuration of that table. RFC 1495 could be described as the process while RFC 1494 is the description of the result; they are really companion documents and should be consulted together.

As described in RFC 1495, MHS messages are comprised of an IPMS.heading and an IPMS.body. The IPMS.body is a sequence of IPMS.BodyParts. An IPMS.BodyPart may be a nested message (IPMS.MessageBodyPart). A MIME message consists of headers and a content. For the purpose of discussion, the content may be structured (multipart or message), or atomic (otherwise). An element of a structured content may be a message or a content. Both message and structured content have subtypes which do not have direct analogies in MHS. The mapping between X.400 and RFC 822 message bodies which the document defines is symmetrical for the following cases:

- any atomic body part
- multipart: digest and mixed subtypes
- message/rfc822

RFC 1327 specifies the mappings for headers. Section 4 describes how those mappings are modified by RFC 1495. Here are the main points when mapping between from an MHS body to MIME:

- Each IPMS.BodyPart is converted to ASCII (since X.400 is not limited to ASCII).

- If the IPMS.Body consists of a single body part, then the RFC 822 message body is constructed as the MIME content corresponding to that body part (as per Table VI).
- If the body part is an IPMS.MessageBodyPart (forwarded IPM), the mapping is applied recursively. Otherwise, the equivalence table is consulted (Table VI) and the specific MHS body part is mapped to the equivalent MIME content-type. If the MHS body part is not identified in the table, then the body part is mapped onto an "application/x400-bp" content.
- If the IPMS.Body consists of more than one body part, then the RFC 822 message body is constructed as a multipart/mixed content-type unless all of the body parts are messages, in which case it is mapped to a multipart/digest content-type. Each component of the multipart content-type corresponds to an IPMS.BodyPart, preserving the ordering of the body parts in the IPMS.Body. There is one case which gets special treatment. If the IPMS.Body consists solely of a single IA5Text body part, then the RFC 822 message body is NOT marked as a MIME content. This prevents RFC 822 mailers from invoking MIME function unnecessarily.

Now from RFC 822 to X.400:

- If an 822.MIME-Version header field is not present, generate an IPMS.Bodypart of type IPMS.IA5TextBodyPart.
- If the 822.MIME-Version header is present, the following mapping rules are used to generate the IPMS.body: If the MIME content-type is one of:
    - any atomic body part,
    - multipart: digest and mixed subtypes, or
    - message/rfc822.

  then the symmetric mapping applies as seen previously. Otherwise, three cases remain: Message/External-body, Message/partial and Nested Multipart Content-types which will not be addressed here.

Now if a user is within an X.400 (1984) network or such a tunnel has to be used, a different process has to be followed. The different process is required since X.400 (1984) did not have externally defined body parts and this is the body part type that will be used for sound, images and video! As seen in **Figure 14**, this is were RFC 1496 [3] comes into play. It proposes rules for downgrading messages from X.400/88 to X.400/84 when MIME content-types are present in the messages. That RFC is also called "HARPOON" which is a pure name that has no meaning. X.400/84 did not have EBPs but X.400 was created with a body that can contain multiple parts of different types. Some body parts are defined by X.400/88 as having both a basic form and an extended form. For all of these, the transformation from the EBP to the basic body part takes the form of putting

the "parameters" and the "data" members together in a "sequence". In the case of the ia5 body part, we would have the extended form:

```
ia5-text-body-part EXTENDED-BODY-PART-TYPE
PARAMETERS IA5TextParameters
     IDENTIFIED by id-ep-ia5-text
DATA            IA5TextData
::= id-et-ia5-text
that is returned to the basic form:
IA5TextBodyPart ::= SEQUENCE{
parameters      IA5TextParameters,
data            IA5TextData}
```

However, this is not possible for EBPs for which there are no equivalent basic body part types. In that case, an encapsulation will take place for each body part. Since the MIME content-types are always in ASCII, it is easier to encapsulate each content-type into an IA5Text body part. The information about the content such as the MIME content-type (jpeg, gif, mpeg, etc) and the content transfer encoding (base64 or quoted-printable) are also encapsulated. This will cause the X.400/84 message to be bigger as the MIME transfer encoding is maintained. However, people with X.400/84 readers who have the ability to save messages to file will now be able to save MIME multimedia messages. If they can identify what to do about a body part, they can now grab implementation of MIME that can run as subprograms and achieve at least some multimedia functionality.

3.5 Encoded Information Types (EITs)

The MIME content-types and their X.400 counterparts, the body part types have been discussed but not the EITs. This is also very important since a given content-type can only be used if its EIT could be identified and understood (this is like compiled information: knowing that an executable is exchanged will not guarantee that it could be run, the recipient must be able to run the given code). The encoding mechanism defined in MIME are: 7bit, quoted-printable and base-64. Information sent in MIME has to be sent in 7bit bytes or encoded in an other form using 7bit bytes. It does not matter what the information is, it could be an image, a sound or a movie clip, it has to be transported in 7bit bytes on lines of a specific length. This is quite limiting and was a significant constraint in the definition of MIME. X.400 does not have such a limitation. The body part types can be sent in any of the "built-in types" defined in ASN.1 (X.208 [21]). ASN.1 is the abstract syntax notation used as a semi-formal tool to define protocols. It is a formal language for describing data structures in a machine-independent fashion. The simple types supported by ASN.1 are found in section one of X.208 while the character string types are at section two. It can be found in the list of types a BitStringType and an OctetStringType. X.400 is not limited to 7bit bytes nor to a fixed line length. The transfer of an arbitrary pattern of an arbitrary length as a string of bits or octets is possible.

3.6 Canonical vs Transfer Encoding

Let us take the example of the image content-type of MIME: the preferred EIT is base64 which was defined in RFC1421 [28]. The canonical encoding will be the one used to encode the pixels on a video screen to the actual computer file. In this case, it can be JPEG or GIF. That encoding will not be touched. It will not be read nor converted in the mapping process. Choosing MIME within X.400 does not require the conversion to a different canonical form as this could take time and can be more or less reliable and lossy. Whether the user is at the end of an X.400 or MIME network, the image will be presented to the user in its native canonical form. This, however, requires the user to have the proper "viewer" for the type received. Again, by selecting MIME types, it is most probable that viewers will be available on most platforms to "view" all the types of information that MIME is transporting. Now the transfer encoding used under MIME (usually base64 for an image) will be removed in the mapping process to X.400 (added to the mapping process to MIME). This requires a conversion that will take time and that may have to be done twice at each end of a tunnel for example. However, unless the X.400 system is of 1984, it is not required and not appropriate to keep the MIME transfer encoding under X.400. The files encoded in base64 grow by one third [34]. In Base64, every 24 bits are exploded into a four-character sequence taken from a special subset of the 7bit ASCII repertoire. The rules are: three octets are taken from the input stream and viewed as 24-bit quantity, which is subsequently divided into four six bit quantities. Each six-bit

value is then indexed into a table of 64 characters. If the input is not an integral number of 24-bit quantities, then an equal-character is used as a pad character for each 8 bit quantity trailing. Hence, a base64 encoding will end with zero, one, or two equal characters. Finally, each line of a base64 encoding must be less than or equal to 78 characters, including the CR-LF sequence. This means that at most an encoding line represents 19 octets from the content value.

4. DESIGN OF THE MULTIMEDIA MODULE

4.1 Environment

The project was conducted in collaboration with a Toronto based company: Zoomit Corporation. Zoomit is working in the X.400 field but not at the UA level. They provide X.400 "engines" on Banyan, Unix and now Windows platforms but they are not involved with the design of UAs. The prototype is built on the PC based Windows environment to benefit from the newly developed Windows X.400 application provided by Zoomit Corp. The Zoomit software is a Windows based application that provides access to an X.400 network through dial-in. A TCP/IP version of the same application is being developed by the company. It will provide the same functionality but through a better connection to the network.

The Zoomit software is providing all the X.400 protocols required for the exchange of E-Mail in one single application. As seen earlier, X.400 has the following protocols:

- P1, the MTS transfer protocol, used between MTAs to provide the distributed operation of the MTS.
- P3, the MTS access protocol, used between a remote user-agent and the MTS to provide access to the MTS abstract service.
- P7, the MS access protocol, used between a remote user-agent and a message store to provide access to the MS abstract service.

We then have the interactions with the elements of interest as in Figure 15. That means that if a user wishes to have access to an MTA directly (P7) or through a message store (P3), it can be done with the same X.400 application. Of course, this all depends on the capability of the UA being used. To exchange multimedia content, it would be required to use the existing protocols and

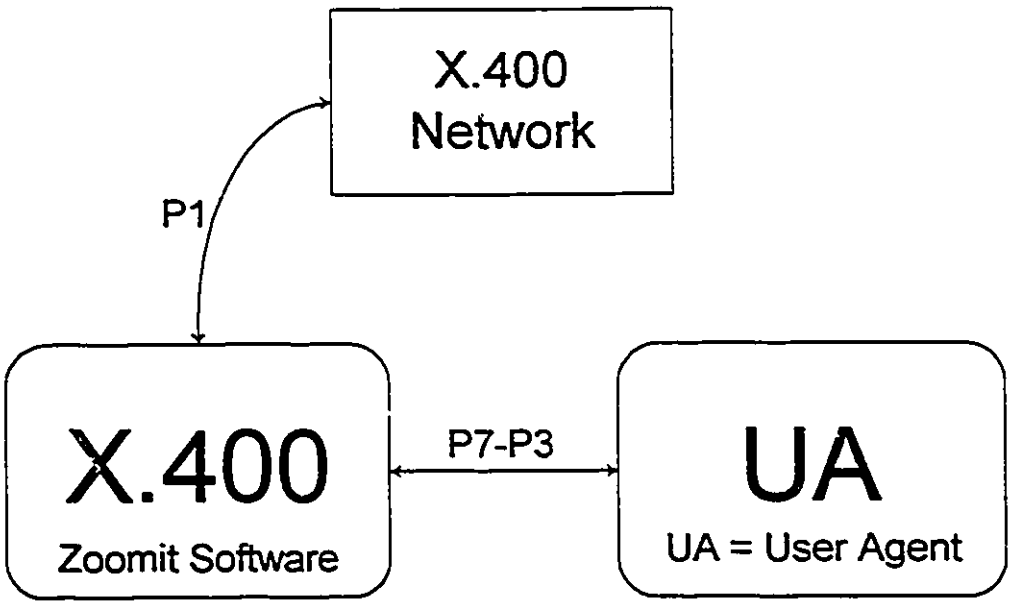

Figure 15 Zoomit Application

channel of communication between the UA and X.400. This has some important disadvantages:

- It is required to modify the standard existing X.400 UAs to enable them to recognize and react to the multimedia content.
- The modification will change the look and feel of the UAs and possibly the way plain E-Mail (text only) is handled by the user.

- The modification is not desirable with regards to security as it combines many different media and different handling mechanisms that may or may not have the same security features.

Instead, the following approach was developed: The multimedia content of a message would be discriminated at the X.400 application level and sent to a different UA that would only handle multimedia information; the Multimedia Display Module (MDM). That approach obviously moves the requirement for

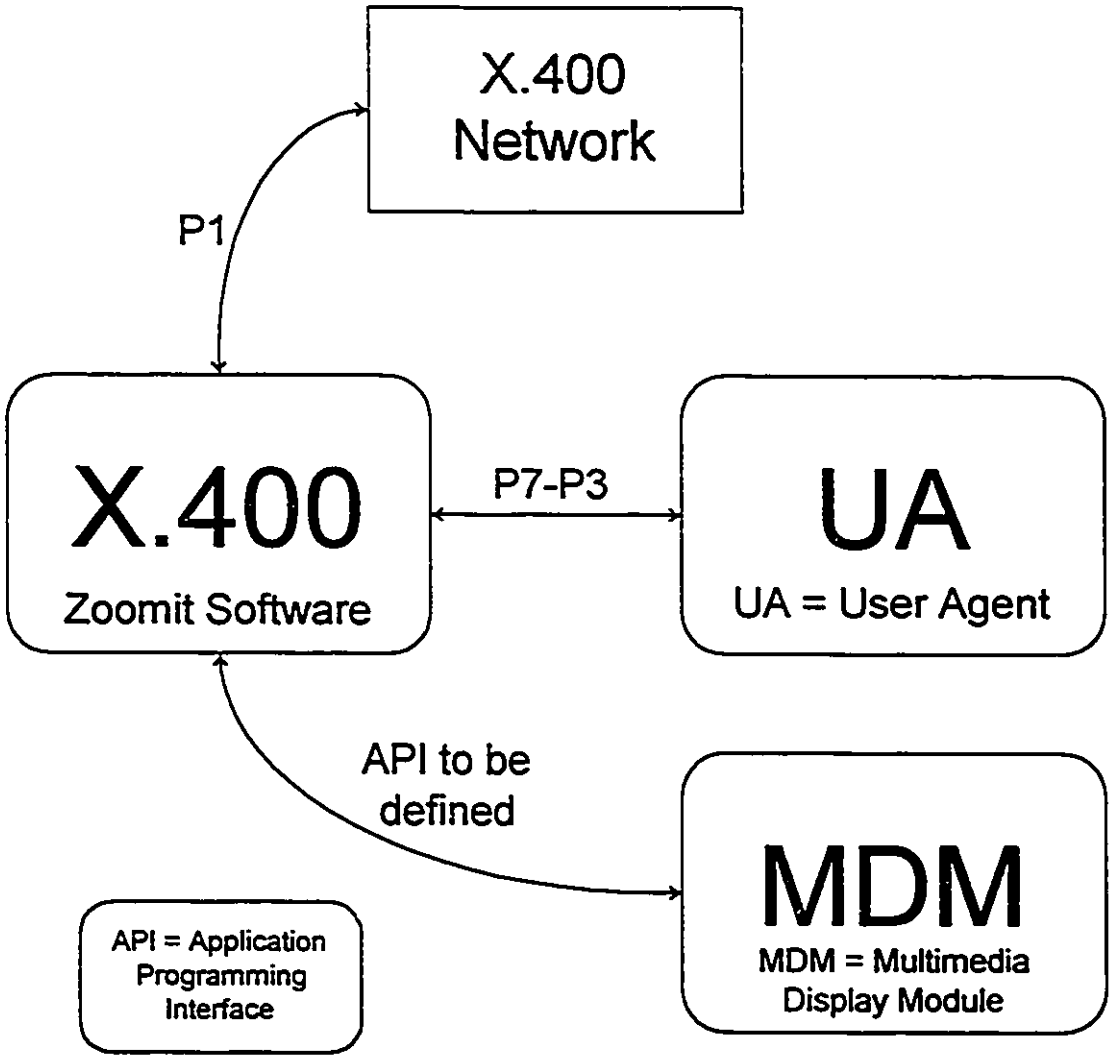

Figure 16 X.400 Application with MDM

modification from the standard UAs to the X.400 application. However, such a modification would not have any effect on a user that does not require or who does not have multimedia capabilities. Such a user keeps the usual UA without any modification in use or look and feel. The separation of text and multimedia content also allows a user to keep a familiar UA for plain text while having a new interface for multimedia content. The diagram of this approach is at **Figure 16.**

4.2 Work Plan

In order to properly use the expertise available and make the best use of the limited time available, it was decided that the modification to the X.400 application would be done by Zoomit while the development of the MDM would be assumed by the author. The boundary between the two would be defined by the API at **Figure 16.**

4.3 MDM Non-Integrated Concept

As it is the case for many MIME UAs, the MDM is not built on an integrated concept. That means that the MDM will not "display[2]" multimedia information as such. It will receive the information from the X.400 application, it will verify the local configuration to see if any given multimedia body part can be handled, it will notify the X.400 application of what can be handled and finally,

---

2 Display is always used in the wide sense: display of images and video and sound reproduction of audio material.

it will provide an interface to the user to trigger the "display" of multimedia body parts (see Figure 17). The MDM is the intermediate between the X.400 application and the local viewers. MDM's task will require a good but flexible configuration mechanism. It has to know the user's system well, pass the information back to the X.400 application properly and launch the appropriate viewer when prompted by the user. The configuration must also be easy to change when new viewers are added for newly defined body parts for example or to reflect a change in viewer installation or user's preference.

4.4 MDM Configuration Mechanism

RFC 1524 [12] is suggesting a file format to be used to inform multiple

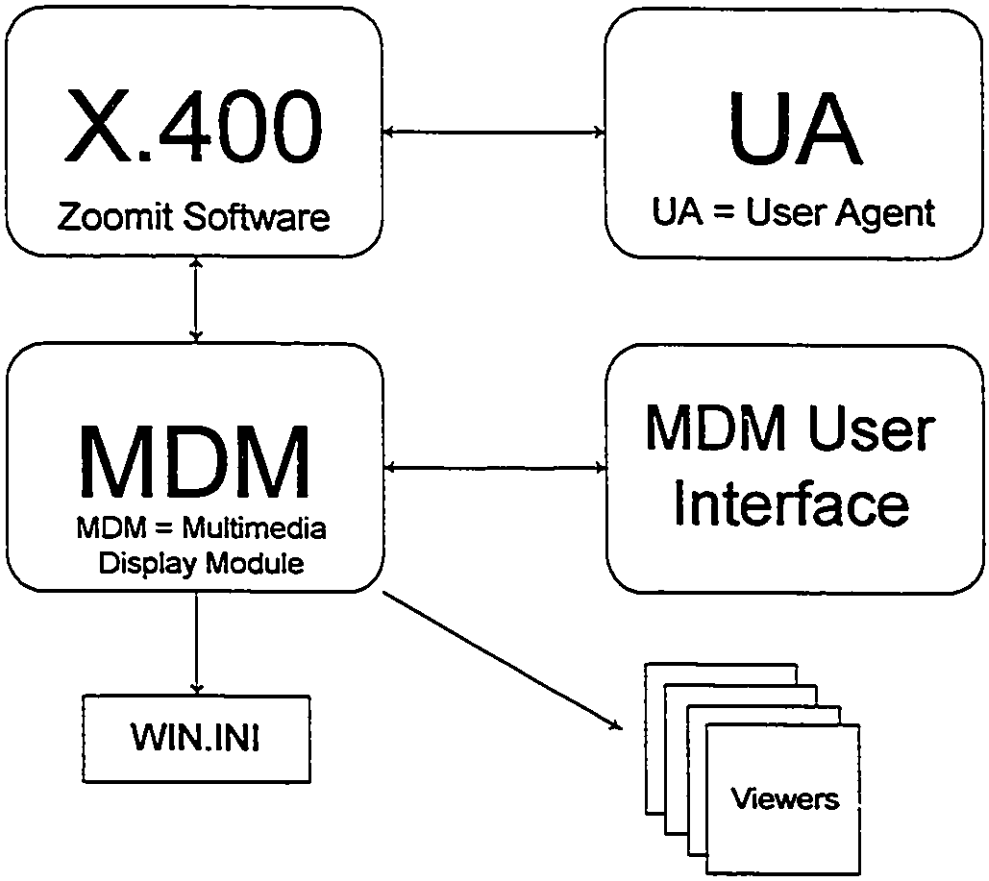

Figure 17 MDM

mail reading user agent programs about the locally-installed facilities for handling mail in various formats. A possible approach is to modify diverse sets of mail reading agents so that, when they need to display mail of an unfamiliar (non-text) type, they consult an external file for information on how to display that file. That approach means that, with a one time modification, a wide variety of mail reading programs can be given the ability to display a wide variety of types of messages. Moreover, extending the set of media types supported at a site becomes a simple matter of installing a binary and adding a single line to a configuration file. "Crucial to this scheme, however, is that all of the UAs agree on a common representation and source for the configuration file." The RFC then proposes such a common representation. It was designed for MIME under the Unix environment where many workstations load the executables from a main server. Now, this is not the situation under study but the same problems will be encountered: Different users at different sites may have different viewers. Users may wish to change the viewer for a given type of body part (or file type). New body parts can be added in MIME so it would be required to support it under this multimedia X.400 concept. Instead of adding a special configuration file (as suggested by RFC 1524) that would have to be maintained, a different method was developed. The method makes use of one of the Windows initialization files (win.ini) and the Windows association feature. That is achieving the purpose of recognizing the system's configuration and user's preferences while providing a

common representation and source for the configuration file. The process is described in more details later.

4.5 Application Programming Interface

The API is an important element of the concept but yet a simple one. It is important because it will be used to ensure proper communications between the two main elements and thus ensuring the proper functioning of the MDM. It is also important because it could lead the way to a standard API that would be used between all developers of X.400 application and MDMs. The use of a

Table VII Required Services Between X.400 and MDM

Service	Supplier	Consumer
OnCallerWindow	MDM	Zoomit X.400
OnRegisterMMBP	MDM	Zoomit X.400
OnSendFileToMDM	Zoomit X.400	MDM
OnZoomitClosed	MDM	Zoomit X.400
OnInquiry	MDM	Zoomit X.400
OnGetFileName	MDM	Zoomit X.400
OnMDMClosed	Zoomit X.400	MDM

standard API would ensure that any X.400 application would work with any MDM. In order to build the API, a list of required services was established as seen in **Table VII.**

OnCallerWindow is the service by which the caller (X.400 application) provide to the MDM a handle to its window while the MDM is returning the handle to its window. As it will be seen, window handles are required in order to exchange messages between applications. The OnRegisterMMBP (Multimedia Body Part) service is how the X.400 application is offering the multimedia body

parts received to the MDM. The return from MDM informs the X.400 application of what body parts can be handled. Then MDM will use OnSendFileToMDM to retrieve each body part as a file. OnZoomitClosed is used by the X.400 application to notify the MDM that the caller (X.400 application) is no longer present. The next two services are required when a multimedia message is composed. OnInquiry will be invoked by the X.400 application to check if the user has placed multimedia contents in the MDM for sending with the message. OnGetFileName is how the X.400 application is requesting the files corresponding to the body parts to be sent. The last one, OnMDMClosed is how the MDM will notify the caller (X.400 application) that the MDM is closing and will no longer be present.

4.6 Construction of the MDM

Three approaches were tested in the construction of the MDM. They are:

- Construction of the MDM as a single Dynamic Link Library (DLL),
- Construction of the MDM as a DLL for the communication and an executable (EXE) for functionality, or
- Construction of the MDM as a single EXE.

Of course, an emulator was also constructed as it was required to emulate the X.400 application's behavior. That module was called the caller application as it is the one calling the MDM to request services. The first approach (Figure 18) was considered as the best approach as far as architecture was concerned. However, the creation of an independent main window in a DLL and its proper

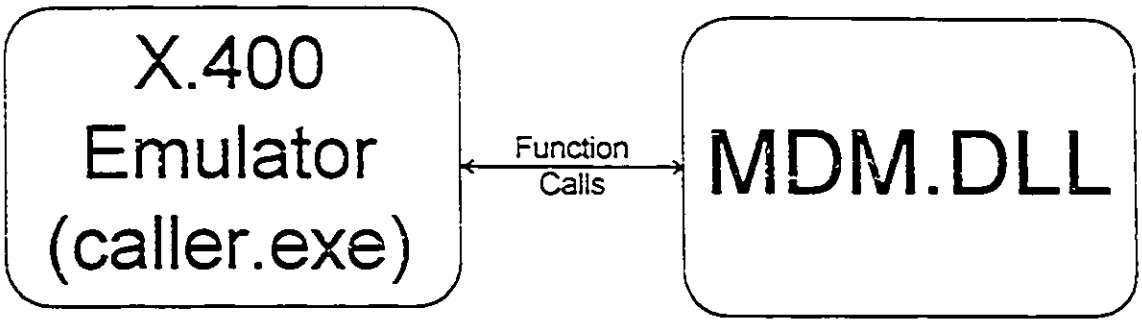

Figure 18 Approach 1

operation has created many problems. The creation of a main window in a DLL is an issue by itself. Actually, various communications over the Internet indicated that it was not possible. But there was no concensus, some said that it could be done without a problem. After weeks of effort, a window was created in a DLL but its operation was quite unstable. Since some operations are initiated by the MDM alone (as a DLL or not), the fact that a DLL is using the calling application's stack generated some concern. Since the operation was still not stable and because of the concerns brought up by Zoomit with regards to the stack, this approach was abandoned.

The second approach was making use of a DLL only as the intermediary between the caller and the MDM. Such a DLL would not have a window nor would it "function" by itself. It would only react to calls from the caller and the MDM and perform the required calls to either of them as appropriate (see Figure 19). This approach was not architecturally as simple as the first one. A new module is introduced that has no purpose other than link the two executables. Its implementation was more complex due to the requirement to export functions from both executables and the DLL and passing the appropriate pointers for

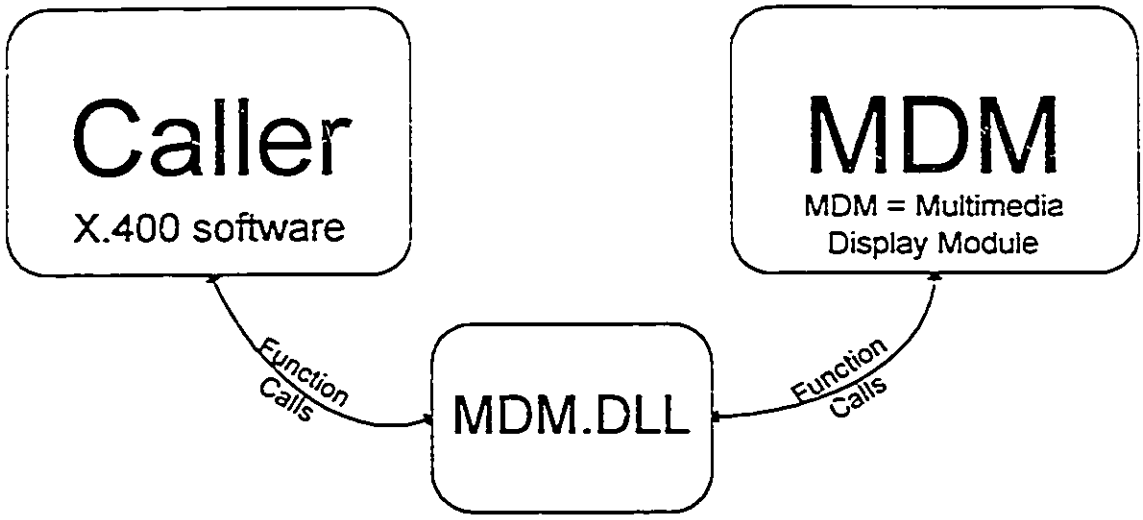

Figure 19 Approach 2

DLL functions to the caller. Such functions are called Callback functions. They are functions within a module that are called back from the outside of it. That procedure was avoided between the DLL and MDM since those two modules could be linked together at compile time. The pointers to the function in the other module are determined and passed at compile time. This is not desirable for the caller and DLL. It is preferable to have the DLL-MDM set independent from the caller so that a given caller (X.400 application) can use the same procedures to access any MDM as long as the API is followed. The MDM could then be changed by the user but the communication between the X.400 application and the new MDM would not be perturbed as long as they both adhere to the API.

The second approach seemed architecturally more complex but still sound. However, the operating system did not allow for such a procedure. When an executable launches or only executes an already launched DLL, the code from the DLL is executed as part of the calling executable task's. In the situation

where the DLL is called by the caller and then executes a callback function in the DLL, that code is still executed as part of the caller's task. Access to data or the C++ "this" pointer then becomes a problem. For example, the buttons in the MDM are created with the "this" pointer. However, the "this" pointer that is accessed even when running in the MDM is the "this" pointer of the caller! There may be convoluted ways to circumvent the problem but it would not be considered a judicious choice. Thus, this approach was also dropped.

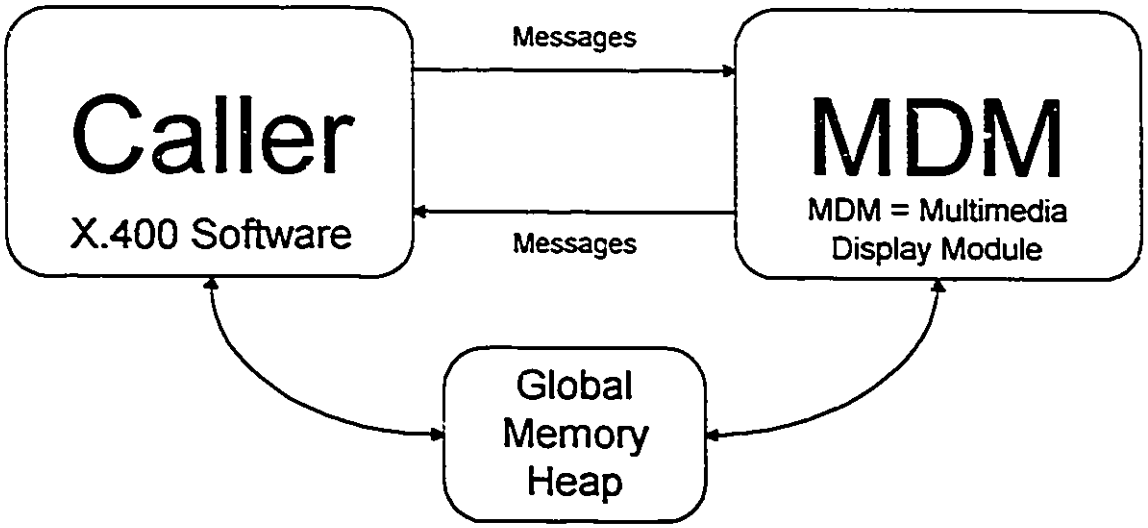

Figure 20 Approach 3

Now, the third and final approach is using message passing between windows (Figure 20). Message passing is a well known method applied to many different environments, why was it not used at the beginning? The payload that messages can carry is very limited: one element of 16 bits and one element of 32 bits. Since we have to exchange sets of pointers, integers, and strings, this methodology was not ideal. Then the issue of passing messages between applications that are not on the same machine is a significant concern. It may be viable for a prototype but this is definitely not appropriate for a commercial

implementation. In most cases, the X.400 application will not be on the same machine as the UA-MDM set. Message passing would not be ideal for that case especially with the access of data in the global heap. A commercial release of the MDM would have to use Dynamic Data Exchange (DDE) or a similar and more flexible process than message passing. Such a process was not used due to the steep learning curve and the limited time to develop the prototype.

Since message passing is used between the two applications, a typical API is not possible as it is usually reserved to function calls. However, to achieve the same goal as a conventional API, the interface between the two applications can be expressed as follows for each message:

Received by:	MDM		
Name:	WM_CALLER_HWND	Value:	5554
Parameters	LPARAM:	X.400 HWND	
	WPARAM:	FAR * to the global structure	
Return	LRESULT:	NIL	
Global Structure	Element	Receiving	Returning
	HWND		MDM HWND
	int		
	UINT FAR*		
	char FAR* FAR*		
	int FAR*		
	UINT FAR*		
	long FAR*		
	const char FAR*		

The other tables for the other messages will be found in appendix 1.

4.7 Microsoft Visual C++

Visual C++ is two complete Windows application development systems in one product [27]. The Visual Workbench is a Windows-hosted interactive development environment that's a direct descendant of Microsoft QuickC for Windows. The Workbench integrates the creation and manipulation of projects. Projects are a collection of interrelated source files that are compiled, linked and bound to make up a working Windows program (somewhat similar to makefiles, projects have the extension .MAK). Visual C++'s second constituent application is the App Studio resource editor. It is an automated tool for the design of Windows resources like dialog boxes and menus. The two constituent elements of Visual C++ with the AppWizard and Class Wizard make a good Integrated Development Environment. AppWizard is a code generator that creates a working skeleton of a Windows application with features, class names, and source code filenames. AppWizard should not be confused with Computer Assisted Software Engineering (CASE) tools. AppWizard is minimalist code; the functionality is inside the application framework base classes. Its purpose is to get the programmer started quickly with a new application. The ClassWizard is a program that operates both inside the Workbench and inside App Studio. ClassWizard takes the drudgery out of maintaining Visual C++ class code. If a function is required to respond to a Windows message, ClassWizard writes the prototype, function bodies, and code to connect the messages to the application framework.

4.8 Microsoft Foundation of Classes (MFC)

The MFC library framework for writing Windows applications simplifies Windows programming by providing the classes needed for user interactions in an application [30]. Most classes in the framework are derived from a single base class at the root of the MFC class hierarchy. This object provides object diagnostics, runtime class information, and object persistence. MFC classes were used as much as possible as they usually provide a much more simplistic approach and much better functionality. For example, the manipulation of strings is much improved with CString as it allocates memory automatically for the string and comes with many "built-in" functions such as to convert to uppercase or lowercase, reverse the string, include up until a given character, etc. It should be noted that MFC is an application framework. The ordinary class library is an isolated set of classes designed to be incorporated into any program, but an application framework defines the structure of the program itself.

4.9 Windows Programs Structure Under MFC

Let's look at an example of a simple Windows program [27]. Here is the header and implementation files for a simple application that displays "Hello, world!":

```
//    application class
class CMyApp : public CWinApp
{
public:
      virtual BOOL InitInstance();
};

//    frame window class
class CMyFrame : public CFrameWnd
{
```

```
public:
      CMyFrame();
protected:
//  `afx_msg' indicates that the next two functions are part of the
//  class library message dispatch system
      afx_msg     void OnLButtonDown(UINT nFlags, CPoint point);
      afx_msg     void OnPaint();
      DECLARE_MESSAGE_MAP()
};
```

And now the implementation file:

```
#include <afxwin>  //  class library header file declares base classes
#include ``myapp.h''  //the above header file

CMyApp NEAR theApp  //  the one and only CMyApp object

BOOL CMyApp::InitInstance()
{
      m_pMainWnd = new CMyFrame();
      m_pMainWnd->ShowWindow(m_nCmdShow);
      m_pMainWnd->UpdateWindow();
      return TRUE;
}

BEGIN_MESSAGE_MAP(CMyFrame, CFrameWnd)
      ON_WM_LBUTTONDOWN()
      ON_WM_PAINT()
END_MESSAGE_MAP

CMyFrame::CMyFrame()
{
      Create(``AfxFrameOrView'', ``MyApp Application'');
}

void CMyFrame::OnLButtonDown(UINT nFlags, CPoint point);
{
      TRACE(``Entering CMyFrame::OnLButtonDown - %lx, %d, %d\n'',
                        (long) nFlags, point.x, point.y);
}

void CMyFrame::OnPaint()
{
      CPaintDC dc(this);
      dc.TextOut(0, 0, ``Hello, world!'');
}
```

The reader used to Windows programming will have noticed that there is no WinMain function because it is hidden inside the application framework. The object of class CMyApp represents an application. When the user starts the

application, Windows calls the application framework's built-in WinMain function, and WinMain looks for the globally constructed application object of a class derived from CWinApp. In C++, global objects are constructed before the main program is executed. When WinMain finds the application object, it calls the InitInstance member function, which makes the calls needed to construct and display the application's main frame window. InitInstance must be overridden in the derived application class because the CWinApp base class doesn't have any information about the type of main frame window that is desired. The usual Run function is also hidden in the base class, but it dispatches the application's messages, thus keeping the application running. WinMain calls Run after it calls InitInstance. The object of class CMyFrame represents the application's main frame window. The ShowWindow and UpdateWindow member functions must be called in order to display the window.

The OnLButtonDown function is an example of the class library's message handling capability. The left mouse button was mapped to a CMyFrame member function OnLButtonDown. The function gets called when the user presses the left mouse button. The OnPaint function is called every time it is necessary to repaint the window: at the start of the program, when the user resizes the window, and when all or part of the window is newly exposed. The CPaintDC statement is a class used to gain access to a device context or in this case the screen. The user shuts down the application by closing the frame window. This action initiates a sequence of events, which ends with the

destruction of the CMyFrame object, the exit from Run, the exit from WinMain, and the destruction of the CMyApp object. The structure found in that example is the structure used in both the MDM and the caller. When AppWizard is invoked to create a new application, it creates a more complex structure that contains documents and views. The document-view architecture separates data from the user's view of the data. The obvious benefit is multiple views of the same data but since this was not required for the applications built, AppWizard was not used to create them.

4.10 The MDM

4.10.1 Flow chart & functionality

A good way to illustrate a Windows program is found in Holzner's book [25]. Using that method, the MDM would be as in Figure 21. The program is "built" around a message loop that is automatically managed by Windows and the MFC framework. Once the program is launched by the calling program (usually an X.400 application), an initialization routine is called from the constructor of the CMainFrame class. The MDM then goes into its message loop and will respond to messages entered in its message map. The first message of interest is the "WM_CALLER_HWND" message. This message is a broadcasted message from the calling program. Upon reception of that message, MDM will obtain the handle to the calling program's window (required to send messages) and respond to the message with its own window handle. Now both applications

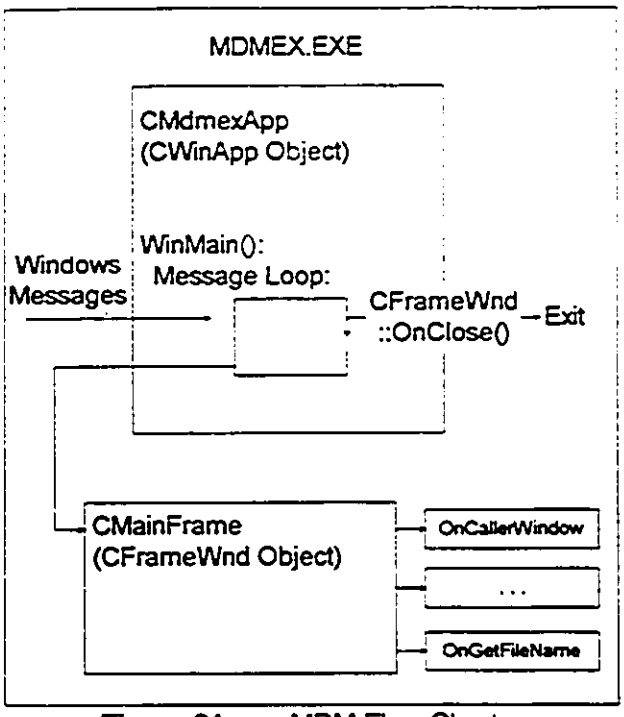

Figure 21 MDM Flow Chart

have each other's window handle and can communicate directly without having to use "broadcast" messages.

Usually, the next message received by MDM will be the "WM_REGISTER" message. With this message the caller is registering a set of received body parts with the MDM. The MDM will call the OnRegisterMMBP function and perform a series of actions. First, it will perform an initialization to clear any existing body parts from a previous message and check if the WM_REGISTER message contained any body parts at all. If zero body parts were sent, the WM_REGISTER message was sent by the calling program just to clear any existing body parts information. That will be used when a message with no multimedia body parts is received after a message that had some.

Once the WM_REGISTER is received, the MDM has a few operations to do (Figure 22). An initialization will be done to clear existing buttons if any, erase the corresponding files and perform data initialization. Then the information about the first body part is obtained from the structure that was globally allocated by the calling application. A test is then performed to find if the recipient system has the capability to display the type of the body part being submitted. If it can be displayed, a decision is made with regards to the type of button that has to be shown, the button is shown in the MDM window and depending if the file is small or large, it is requested to the calling

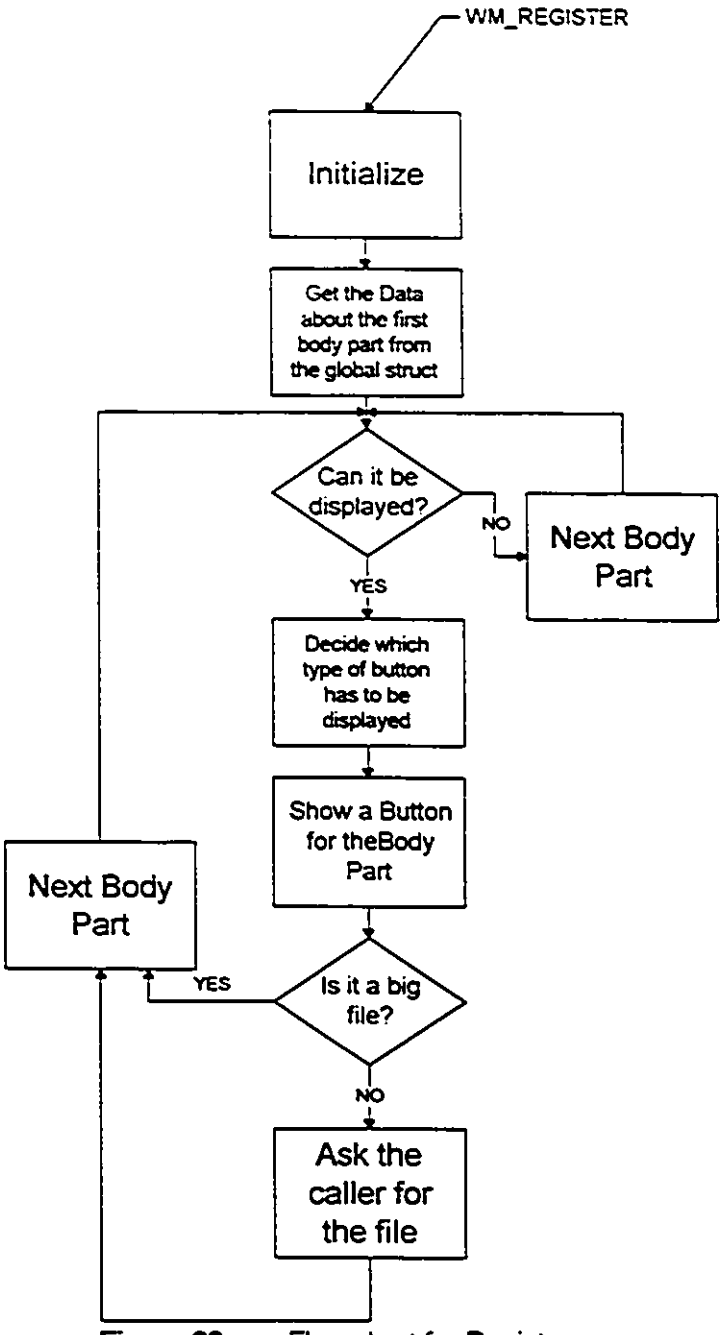

Figure 22 Flow chart for Register

application (small) or not (large). This alleviates the problem of having the system busy transferring a large file if the recipient does not wish to see it at that time. The process is repeated for each body part submitted and when a body part is "accepted" by the MDM, a body part count is incremented and the corresponding sequence number is returned to the calling application through the global structure.

OnSendFileToMDM is the function used by the MDM to request that the calling application send a body part to the destination requested by MDM. It is called immediately after a register for small body parts and upon request by the recipient for large body parts. OnZoomitClosed is used by MDM to take note that the calling application has closed. OnInquiry is used by the calling application to inquire to the MDM if there are body parts waiting to be sent. This will be used when the recipient wishes to send multimedia body parts with a message being composed with the text UA. Once the send command is received by the X.400 application it should inquire to the MDM for multimedia body parts. If there are some, the X.400 application will then use OnGetFileName to obtain the filenames and locations of the files that have to be loaded and sent as multimedia body parts. OnMDMClosed is used by the X.400 application to take note that the MDM has been closed.

4.10.2 Classes & Objects

The MDM is made of only two objects as in the example shown previously: one object of class CWinApp (CMdmexApp) for the application itself and one

object of class CFrameWnd (CMainFrame). All of the functionality is built in the CMainFrame object. A few other MFC classes are used as elements of the MDM. The major one is the CAddBodyPart class derived of the CFileDialog base class. That class (by its declared object "dlg") is used in the Add Body Part menu entry for the selection of a file that will be included in the MDM window. That class/object provides a standard file selection and navigation interface common to almost all Windows applications. Other objects of significant importance are the CBitmapButtons, the CStrings and CStringArrays.

The buttons displayed in the MDM window are CBitmapButtons objects. Since they are created dynamically, they are created in the following manner:

```
CBitmapButton * gButtonArray[MaxNumberOfButtons];
gButtonArray[current_button_count] = new CBitmapButton;
gButtonArray[current_button_count]->Create(``Button Text'', PROPERTIES,
                    SIZE, ParentWindowHandle, ButtonIDNumber);
gButtonArray[current_button_count]->LoadBitmaps(Type_of_Button_UP,
                          Type_of_Button_DOWN);
```

It is then possible to create and delete the objects without having to manipulate names. CStrings behave like normal strings in most cases but have built-in functions that are quite useful. CStringArrays are arrays of CStrings. It is interesting to note that once an object is declared as a CString, it is not required to allocate memory for the actual strings. The object will allocate and deallocate memory automatically depending on the strings that it contains. It is the same for the CStringArray except, of course, that the size of the array has to be explicitly given. The DecideButton function has a good example of the use of

CString functions such as MakeUpper, MakeReverse and SpanExcluding. CStrings were used wherever possible instead of usual strings.

4.10.3 Functions

The MDM (CMainFrame) has the following functions:

- Initialize, for the initialization described above;
- CheckAssociation, for checking if a body part type can be displayed by the recipient;
- DecideButton, to decide which button type to display;
- ShowButton, to create the buttons in the MDM window; and
- LoadFile, to ask the calling application to send a body part to the given location under the given name.

However, it also has 12 message and command handling functions which are making an important part of the MDM. They are:

- OnClose, OnQueryEndSession, OnHelp, OnFileNew and OnFileExit, the usual Windows functions adapted for the MDM;
- OnFileAddBodyPart, for the addition of multimedia body parts in compose mode;
- OnLaunch for the launching of a body part viewer;
- OnCallerHwnd, to accept obtain the calling application's window handle and to return the MDM's window handle;

- OnRegisterMMBP, to register the body parts submitted by the calling application;
- OnZoomitClosed, to take note of the closing of the calling application;
- OnInquiry to provide the number of multimedia body parts in the MDM window to the calling application; and
- OnGetFile, to provide the names of the files corresponding to the multimedia body parts number sent as a reply to OnInquiry.

Most of those functions are small and easy to follow. OnLaunch will be discussed as it is a more important function (**Figure 23**) and

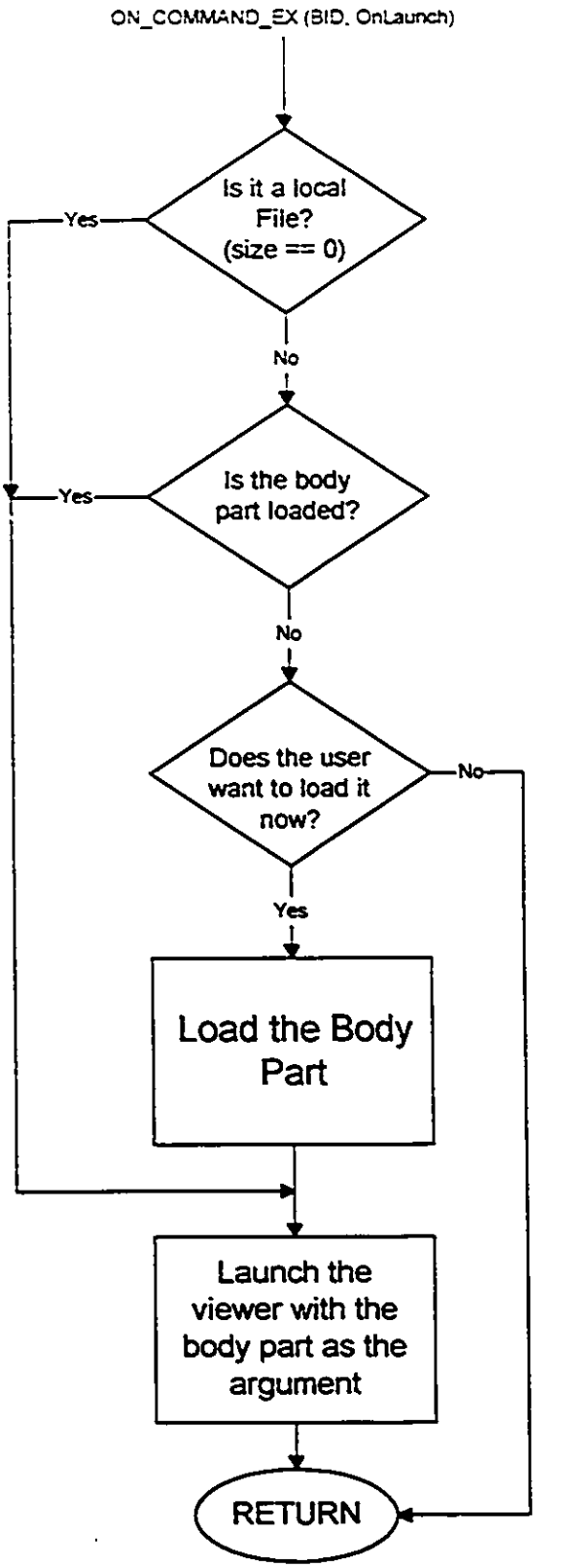

Figure 23 OnLaunch Flow Chart

CheckAssociation as it contains the mechanism that prevents the requirement for a configuration file maintained by MDM.

Since a body part may or may not have been loaded when the button was first displayed in the MDM window, the OnLaunch function must verify if the file was loaded or not but also if it is a local file or not. In the case of local files, they are always loaded because the recipient has access to them locally. A file size of zero will indicate that the file is local. In the future, it might be interesting to obtain the actual file size and to pass it on to the calling application. The behavior of that calling application may be different depending on the size of any given body part. So, if the file is local or if the file was previously loaded, the corresponding viewer is launched with the body part as the argument. If it was not loaded, the user is offered the choice of loading it now or not. In the latter case, the function returns without loading nor displaying the body part.

The CheckAssociation function has one line of code that provides the same functionality of an MDM managed configuration file:

```
numberbytes = GetProfileString(``Extensions'', Type, ``'', pszBuf, 80);
```

That line is using a standard Windows command to verify in the WIN.INI file if an extension of the type "Type" is associated with an executable file. For example, this is the mechanism that allows a user to double-click on a TXT file and automatically launch the Windows notepad with the double-clicked file loaded. Now, that imposes some work upon the user: every time the user installs

or changes his preference for a viewer of files of a specific type. That user must then use the Windows File Manager to select a file of the specific type and then click on File - Associate and then select a viewer. That viewer will now be used whenever a file of the specific type is double-clicked or launched by the MDM. The work imposed on the user is not greater that the work required to maintain a separate MDM configuration file but it makes the maintenance of that configuration standard and much simpler.

4.11 The X.400 Application Emulator

4.11.1 Flow chart & functionality

The X.400 application emulator (the Caller) has the same structure as the MDM (**Figure 24**). As seen previously, Caller will only respond to two messages sent by the MDM: WM_MDM_CLOSED and WM_SEND_TO_MDM. The first one is only for the caller to take note that the MDM was closed. Subsequent calls such as register will require the caller to launch a new MDM. Or, in the event that the caller is also closed, it would not have to notify the MDM as the latter is already closed. As the caller is only meant to be an emulator, it only performs the basic

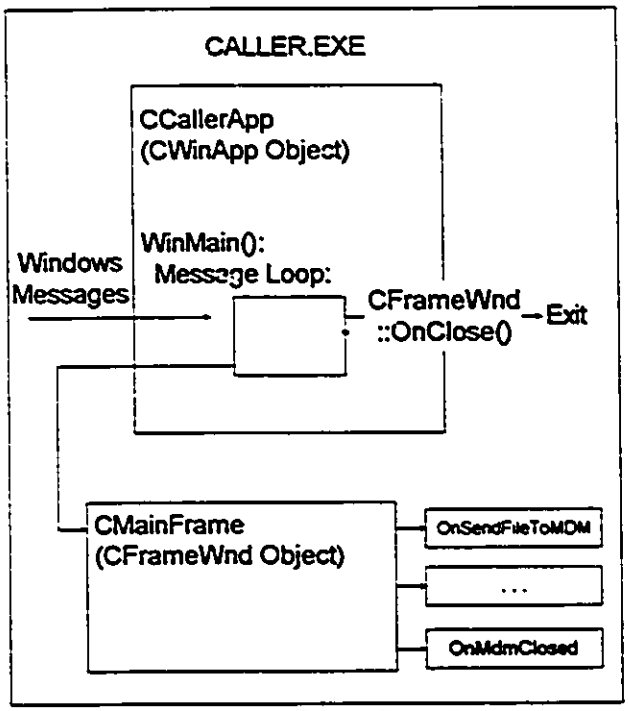

Figure 24 Caller Flow Chart

calls to the MDM to ensure that the MDM is properly functioning. The caller has menu items for "Register, Inquire, and Get file".

Once Register is clicked, the caller will launch the MDM, if required, and send a WM_CALLER_HWND broadcast message. That will provide to the MDM the window handle of the caller and the caller will get in return the window handle of the MDM. Then the global structure is loaded with parameters and the WM_REGISTER message is sent to the MDM.

The caller will then return to the message loop and wait for new commands or messages. Usually, a set of WM_SEND_TO_MDM messages will be received for the small body parts. The caller will only copy the files from a defined subdirectory to the location provided by the MDM. With the reception of a WM_MDM_CLOSED message, a flag will be set to ensure that no communications is attempted with the MDM.

4.11.2 Classes & Objects

The caller is also made of only two objects and all of the functionality is built in the CMainFrame object.

4.11.3 Functions

The Caller (CMainFrame) has the following functions:

- InitializeRegisterData, for the initialization of the test register data and relevant variables;
- PrepareStruct, to make the parameter passing structure globally available;

- copyfile, to emulate the transfer of a body part to a file as requested by the MDM; and
- NotifyMDM, to notify the MDM that the caller is closed.

However, it also has nine message and command handling functions which are making an important part of the caller. They are:

- OnClose, OnQueryEndSession, OnHlp and OnFileExit, the usual Windows functions adapted for the caller;
- OnRegister, called when the user hits the Register menu item, to register the test multimedia body parts with the MDM;
- OnInquire, called when the user hits the Inquire menu item, to inquire to the MDM about the number of body parts available for sending; and
- OnGetFile, called when the user hits the Get File menu item, to ask the MDM for a file name and location for a given body part.

The PrepareStruct function has an interesting element in the global allocation of the structure:

```
hgStruct = GlobalAlloc(GMEM_SHARE, sizeof(struct ParameterList));
lpStruct = (struct ParameterList FAR *)GlobalLock(hgStruct);
```

That ensures that the data placed in the structure is available to the MDM (as well as to all other applications running). Each element of the structure is then accessed using the usual pointer notation (->).

5. RESULTS

5.1 Testing Environment

In the testing environment, the MDM behaved as expected. Once the register menu item was selected in the caller application, the MDM appeared as in Figure 25.

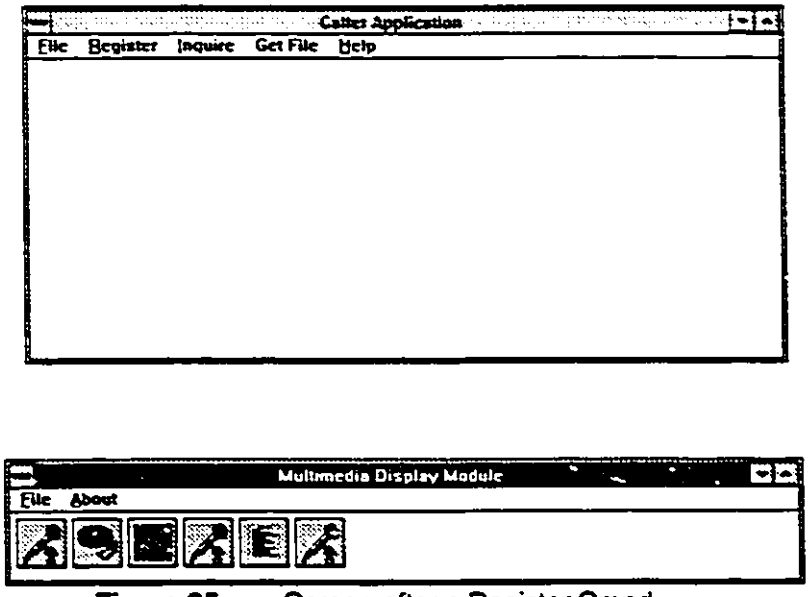

Figure 25 Screen after a Register Cmnd

Once a button is hit in the MDM, the corresponding body part is shown with the associated viewer (Figure 26). Depending on the viewer associated with each type of body part, it is then possible not only to view or hear a body part but also to save the file containing the multimedia body part wherever desired. Figure 27 illustrates the saving process for an image file.

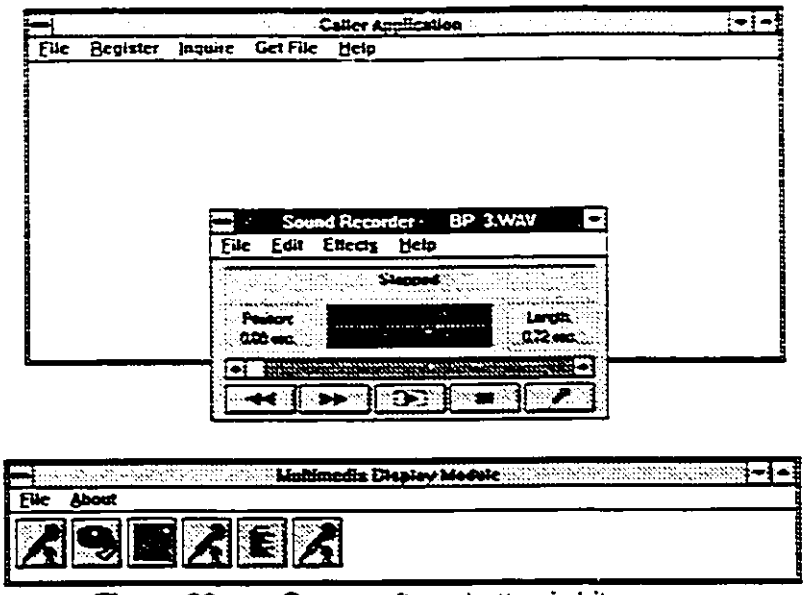

Figure 26 Screen after a button is hit

The MDM interface is simple but still intuitive. If viewing a body part is desired, the button corresponding to it only has to be hit. If, after seeing the body part, the user desires to save it, it can be done by using the usual features of the associated viewer. The procedure for saving is not changed. The temporary files created by the X.400 application with the names provided by the MDM are

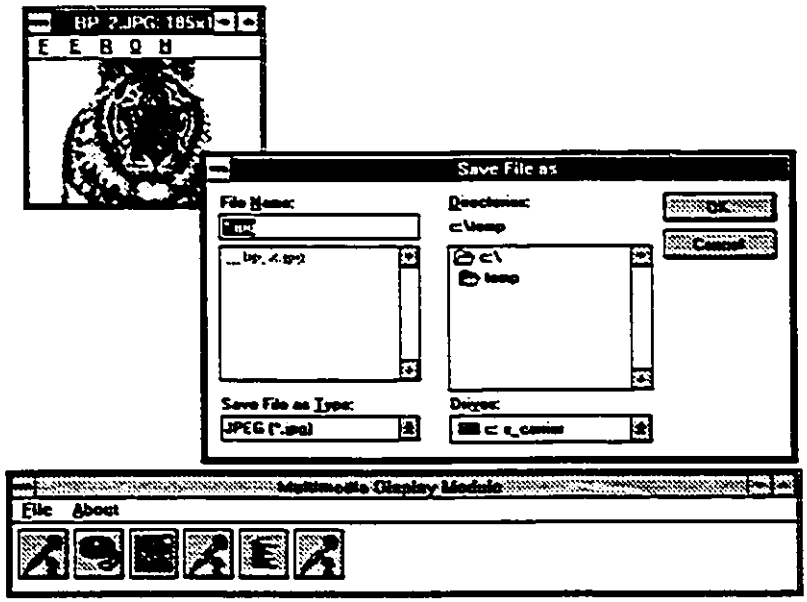

Figure 27 Saving Process

automatically erased when a new Register is received or when user selects File-New.

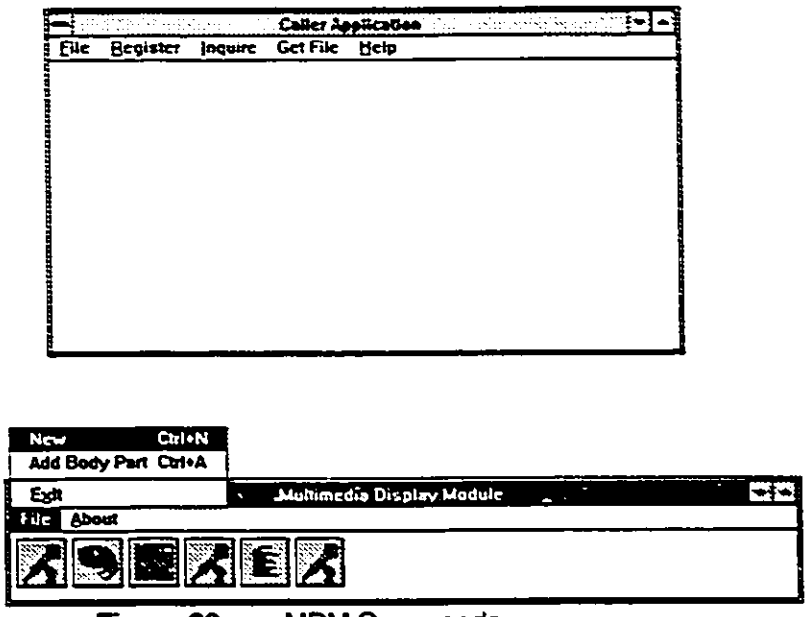

Figure 28 MDM Commands

The user has access to very few commands in the MDM as it can be seen in Figure 28. However, those commands allow the user to receive and send multimedia body parts without complications. When multimedia body parts are

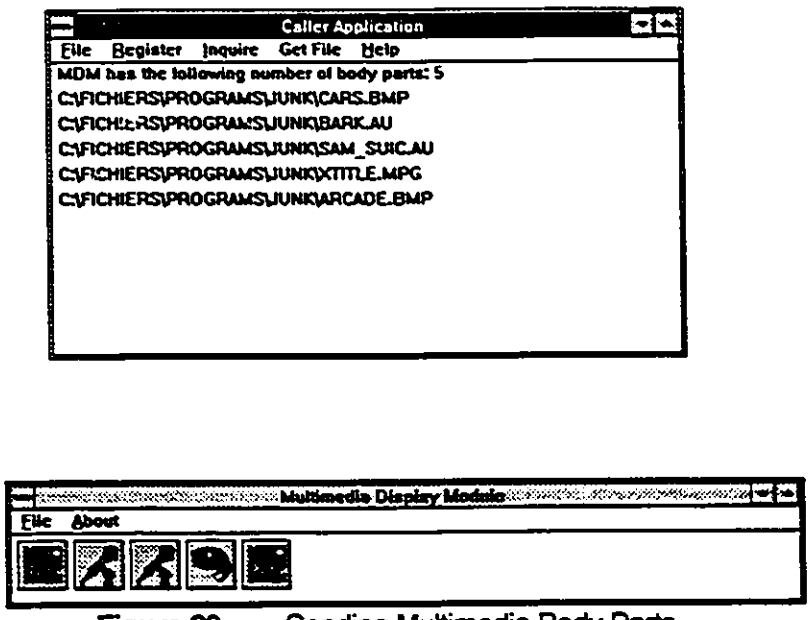

Figure 29 Sending Multimedia Body Parts

received, the user hits a button to see and save the body part. For sending multimedia body parts, the user selects the Add Body Part command and buttons will appear as each body part is added.

For example, in Figure 29 the user has selected five body parts and the caller has inquired the MDM about multimedia body parts to be sent. That inquiry would be done when the user hits the send button on his UA.

In the case that multimedia body parts were registered with MDM but not loaded on the user's station, a message will be issued when the corresponding button is clicked (Figure 30). If the user hits cancel, the body part

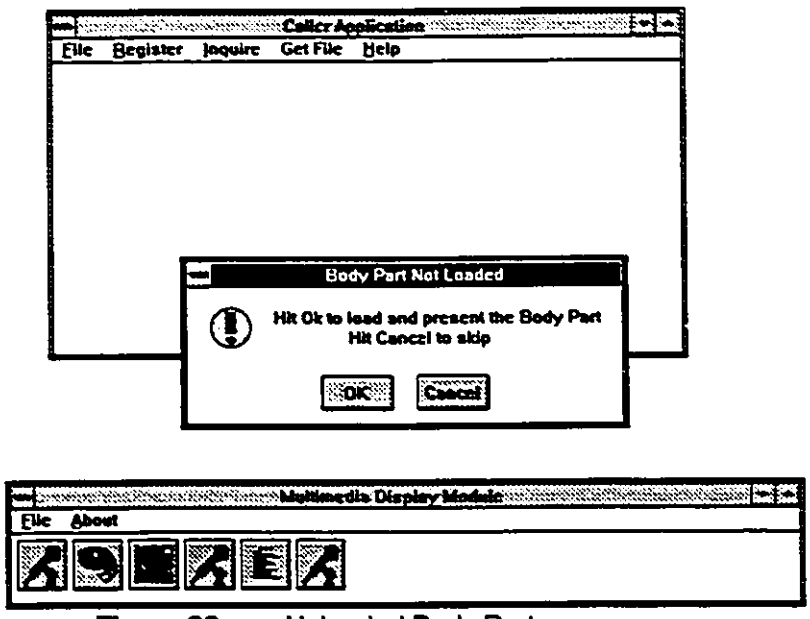

Figure 30 Unloaded Body Part

is not viewed nor is it loaded. If the user hits OK, a message is sent to the caller requesting that the multimedia body part be sent to a given location with a given file name (file name with path). The body part is then viewed as if the button

was hit for a loaded body part. Subsequent hits on the same button obviously do not generate the same message as the body part was already transferred.

5.2 Real Environment

Since the software was still not available from Zoomit at writing time, only a screen simulation of the real environment was done. Figure 31 shows how the MDM would look with a real UA on a screen. Functionality will be the same but no real messages were tested.

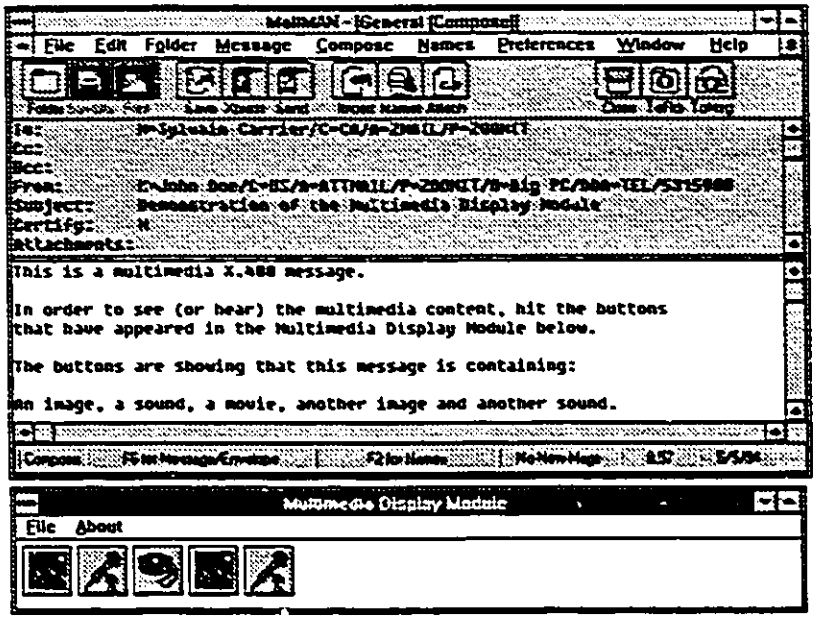

Figure 31 MDM with a real UA

6. CONCLUSION

The goal of this research was to find a way to provide multimedia mail on X.400 in a compatible, interoperable and useable fashion. There may be an increased demand for multimedia mail but an implementation must be compatible with the existing X.400 standard to prevent any problems using the infrastructure already in place. In addition to be compatible, it must also be interoperable with the others. Since the others form the majority in North America with the Internet mail, it is imperative to be seamlessly interoperable with the Internet mail. All is not well when those two goals are achieved: the resulting concept must be useable. It must not create a situation in which a user would have the capabilities desired only at an extreme cost with regards to ease of use and simplicity. These requirements look much like the goals that were set for the MIME effort.

This effort has been made in collaboration with an industrial partner: Zoomit Corporation of Toronto. The collaboration was meant to be a transfer of technology to the industry but also a means to obtain more complete results; once Zoomit has included the work of this thesis we will have a complete and functional X.400 application from user to user. However, that has implied that the research had to include the complete user interface in the Windows environment and the corresponding programming. Zoomit would then have been the only one manipulating the message handling side and their proprietary X.400 application.

A brief introduction to X.400 was given and the most important elements of the recommendation were described. That introduction should have been enough to realize that X.400 is an exhaustive and complex standard that should not be changed. Changes to a complex standard can create protocol conflicts but the process to have an international standard changed is also complex or at least quite long.

Then a decision process to select a solution is discussed. Four options are analyzed: use a proprietary format, use new body parts, use an encapsulation of MIME message or use a mapping process of MIME content-types. The proprietary format would easily integrate the authoring process (compose/display) the interchange format and even the logical synchronization of all the multimedia elements. However, it would basically defeat the purpose of the standard: using a proprietary format that includes - and hides - everything is far from being an open concept to facilitates the exchange of E-Mail between countries, companies, etc.

Then the option of selecting new body parts could offer a lot but with significant drawbacks. The new body parts selected could be optimal for a given media and offer very good performance. However, those would have to be introduced in the standard after an agreement has been reached. Now, this is certainly not done quickly but there is more. The new body parts selected would now be the standard on X.400 and would solve the interchange format. However, the standard tools to manipulate them might not be available for some time and

worse, those new body parts may not be compatible with the Internet and the formats available now.

To ensure interoperability with the Internet, there is nothing better than using the Internet (MIME) format. MIME messages can be encapsulated as a whole or each of their content-types can be mapped to MIME-equivalent X.400 body parts. Encapsulation could be done relatively easily: a MIME message is composed by the user, encapsulated in a given X.400 body part and sent on the X.400 network. Interoperability is ensured and the composers/readers used would be MIME readers/composers. That would be a quick and dirty solution but it has problems. By using a MIME message encapsulated, we introduce a foreign format in X.400. MIME may not be proprietary but it is not an X.400 format either. That creates a functionality problem within X.400: if an encapsulated MIME message is received and the recipient is not equipped to properly "read" it, the recipient's UA may be lost and may not even provide the text portion. That situation could even be called an interoperability problem within X.400. Non-MIME ready X.400 users would see an enormous text file without easily knowing what is inside. Also, that encapsulated MIME message would have retained its transfer encoding and thus causing a significant increase in size. Multimedia files are big enough as it is, that alone could be a reason not to use this solution.

The option selected was to use the MIME content-types as independent X.400 body parts. That does not require a change in the standard because no

new body parts will be created. Each MIME equivalent body part will be carried by an X.400 externally defined body part (or extended body part). There is an effort required to identify the MIME content-types in a uniform and standard way but the content itself can only be the same as the MIME content-types. The content will suffer no canonical encoding conversion but only a transfer encoding conversion. That ensures that the body parts are as small as possible but also that MIME viewers can actually be used at both ends since the canonical form is preserved. A non MIME ready X.400 user can now have access to the message and discriminate easily each body parts. That non-MIME ready user may not be able to view all the body parts but it will be easy to determine what the message contains.

At least two implementation approaches were possible: build an integrated MIME enabled X.400 UA or build a separate module that will only handle the multimedia body parts. The latter was selected because it maximizes the usability concept: a message without multimedia content will cause no change in the user's procedure. In that case, the same UA can be used in the same way as previously. Also, the standard commercial UAs do not have to be changed, patched or adapted. However, the software providing the X.400 service to the UA will have to be changed to communicate with the multimedia display module. This is not the approach selected by the major projects such as BERKOM and Eurobridge. They also use externally defined body parts to carry the multimedia information and thus are compatible with this approach.

However, in the case of BERKOM for example, their aim was to provide the service for composing, sending and receiving multimedia messages as well as a framework for their presentation. This research has not followed that approach in order to capitalize on the users' knowledge as much as possible, remain very useable and maximize user acceptance

The multimedia display module would then be the interface between the X.400 software and the MIME viewers. The MIME content-types offer standardization where X.400 was lacking but they also offer body part types that can be viewed using MIME viewers. The design and the implementation of the multimedia display module was described in detail. It offers a simple and effective interface to the user that shows the multimedia content of a message at a glance and allows multimedia files to be selected for outbound messages.

The configuration mechanism for the multimedia display module follows the concept proposed by the creator of MIME but implements it by making good use of the Windows environment. The multimedia display module does not have to maintain a configuration file as it is done by the Windows operating system. Yet it is easily accessible and an intuitive Windows procedure.

The concept chosen is not a quick and dirty solution but rather a pragmatic approach to multimedia mail on X.400. It is not revolutionary but makes a good use of what is available to produce a simple and effective method that produces multimedia electronic mail on the X.400 message handling system.

Bibliography

[1] Alvestrand H., and S. Thompson, "Equivalences between 1988 X.400 and RFC-822 Message Bodies", *RFC-1494*, 19p, (August 1993).

[2] Alvestrand H., S. Kille, R. Miles, and S. Thompson, "Mapping between X.400 and RFC-822 Message Bodies", RFC-1495, 11p, (August 1993).

[3] Alvestrand H., J. Romaguera, and K. Jordan, "Rules for Downgrading Messages from X.400/88 to X.400/84 when MIME Content-Types are Present in the Messages", RFC-1496, 5p, (August 1993).

[4] Alvestrand H., Private communications, (26, 29 Nov 1993).

[5] Armbrüster H., and Wimmer K., "Broadband Multimedia Applications Using ATM Networks: High-Performance Computing, High Capacity Storage, and High Speed Communication", IEEE Journal on Selected Areas in Communications, vol. 10, no 9, (December 1992), pp. 1382-1396

[6] Baveco M.P.P. et al., "MultiMedia Mail using OSI: Bridging the gap between what exists and what is required", Second International Conference, Broadband Islands, Athens, (June 1993), pp. 193-196.

[7] Borenstein N. and Freed N., "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", RFC 1521, (September 1993).

[8] Borenstein N.S., "Multimedia Electronic Mail: will the dream become a reality?", Communications of the ACM, vol. 34, no 4, (April 1991), pp. 117-119.

[9] Borenstein N.S. & Thyberg C.A., "Power, ease of use and cooperative work in a practical multimedia message system", International Journal Man-Machine Studies, vol. 34, (1991), pp. 229-259.

[10] Borenstein N.S., "MIME: a portable and robust multimedia format for Internet mail", Multimedia Systems, vol. 1, (1993), pp. 29-36.

[11] Borenstein N.S., "Internet Multimedia Mail with MIME: Emerging Standards for Interoperability", Proceedings of the IFIP WG 6.5 International Symposium on Upper Layer Protocols, Architectures and Applications, (1992), pp. 183-192.

[12] Borenstein N.S., "A User Agent Configuration Mechanism For Multimedia Mail Format Information", RFC 1524, (September 1993).

[13] Bozios T. & Pronios N.B., "Multimedia Synchronisation: The Role of the Communication System", Second International Conference, Broadband Islands, Athens, (June 1993), pp. 91-102.

[14] Brachman B.J. and Chanson S.T. "A Simulation Study of Application Level Message Transfer Using Message Streams" Computer Networks and ISDN Systems, vol. 19, (1990), pp. 79-94.

[15] Carrier S., Georganas N.D., "Multimedia Electronic Mail on X.400: A Pragmatic Approach", Proceedings of the 5th IEEE Communications

Society International Workshop on Multimedia Communications. Kyoto, Japan, (16-19 May 1994).

[16] Communications Week, "Washington Watch", (16 May 1994), p43.

[17] Edwards W.K., "The Design and Implementation of the MONTAGE Multimedia Mail System", Proceedings of IEEE TRICOMM 1991, pp. 47-57

[18] Hand S., "Towards the Messaging System of Tomorrow", Second International Conference, Broadband Islands, Athens, (June 1993), pp. 189-192

[19] Hardcastle-Kille S., "Mapping between X.400(1988) / ISO 10021 and RFC 822", Request for Comments (RFC) 1327, 113 p, (May 1992).

[20] Huitema C., "The Challenge of Multimedia Mail", Computer Networks and ISDN Systems, vol. 17, (1989), pp. 324-327.

[21] ITU-CCITT Blue Book, Volume VIII - Fascicle VIII.4, "Data Communication Networks Open Systems Interconnection (OSI)", Recommendation X.200-X.219, IXth Plenary Assembly, Melbourne, (14-25 November 1988).

[22] ITU-CCITT Recommendation X.420 - Data Communication Networks-Message Handling Systems - Interpersonal Messaging System, (09/92).

[23] ITU-CCITT Blue Book, Volume VIII - Fascicle VIII.7, "Data Communication Networks Message Handling Systems",

Recommendations X.400-X.420, IXth Plenary Assembly, Melbourne. (14-25 November 1988).

[24] Hofrichter K., Moeller E., Scheller A. And Schürmann G., "The BERKOM Multimedia-Mail Teleservice", Proceedings of the 4th Workshop of future trends in distributed communication systems, Lissabon, (22-24 April 1993), Springer Verlag.

[25] Holzner S., "Visual C++ Programming", Brady Publishing, New York, (1993).

[26] Kretz F. & F. Colaïtis, "Standardizing Hypermedia Information Objects", IEEE Communications Magazine, vol. 30, no. 5, (May 1992), pp. 60-70

[27] Kruglinski D.J., "Inside Visual C++", Microsoft Press, Redmond, (1993).

[28] Linn J., "Privacy Enhancement for Internet Electronic Mail: Part 1, Message Encryption and Authentication Procedures", RFC 1421, 42p, (February 1993).

[29] Naffah N. & A. Karmouch, "Agora - An Experiment in Multimedia Message Systems", IEEE Computer Magazine, vol. 19, no 5, (May 1986), pp. 56-66.

[30] Pappas T.L., "MS Visual C++: A Simplified Approach", IEEE Computer Magazine, vol 27, no 1, (January 1994), pp 85-88.

[31] PC Magazine, "Trends in Brief", vol 13, no 11, (14 June 1994), p32.

[32] Plattner B., C. Lanz, H. Lubich, M. Müller, and T. Walter, X.400 Message Handling - Standards, Internetworking, Applications, Addison-Wesley, Wokingham, England, (1991).

[33] Radicati S., Electronic Mail - An Introduction to the X.400 Message Handling Standards, McGraw Hill, New York, (1992).

[34] Rose M.T., The Internet Message - Closing the Book with Electronic Mail, Prentice Hall, Englewood Cliffs, (1993).

[35] Stallings W., ISDN and Broadband ISDN, Macmillan Publishing Company, New York, (1992).

[36] Stefferud E., "Interworkability - Looking Back From the Future", Network Computing, vol. 4, no 13, (November 1993).

[37] Tanenbaum A.S., Modern Operating Systems, Prentice Hall Inc, Englewood Cliffs, (1992).

[38] Technology in Government, "Snapshot", (May 1994), p 11.

[39] Thomas R.H. et al. "Diamond: A Multimedia Message System Built on a Distributed Architecture", IEEE Computer, vol. 18, no. 12, (December 1985).

[40] Venkat Rangan P., Kaeppner T. and Vin H.M., "Techniques for Efficient Storage of Digital Video and Audio", Proceedings of 1992 Workshop on Multimedia Information Systems (MMIS '92), (February 1992), pp. 68-85.

[41] Yamamoto B., "Multi-media electronic mail", Proceedings of SPIE vol. 1258 Image Communications and Workstations, (1990), pp.2-9.

Appendix 1 - Application Programming Interface

This appendix contains a set of tables for each message exchanged between the X.400 application and the Multimedia Display Module.

Received by:	MDM		
Name:	WM_CALLER_HWND	Value:	5554
Parameters	LPARAM:	X.400 HWND	
	WPARAM:	FAR * to the global structure	
Return	LRESULT:	NIL	
Global Structure	Element	Receiving	Returning
	HWND		MDM HWND
	int		
	UINT FAR*		
	char FAR* FAR*		
	int FAR*		
	UINT FAR*		
	long FAR*		
	const char FAR*		

Received by:	MDM		
Name:	WM_REGISTER	Value:	5556
Parameters	LPARAM:	NIL	
	WPARAM:	NIL	
Return	LRESULT:	NIL	
Global Structure	Element	Receiving	Returning
	HWND		
	int	NUMBER OF BP	
	UINT FAR*	SEQUENCE NOs	
	char FAR* FAR*	TYPES	
	int FAR*	* MDM BP HANDLED	
	UINT FAR*	* MDM SEQ NO HANDLED	
	long FAR*	* MDM BODY PART SIZE	
	const char FAR*	* MDM FOR FILE NAME	

Received by:	MDM			
Name:	WM_ZOOMIT_CLOSED		Value:	5559
Parameters	LPARAM:	NIL		
	WPARAM:	NIL		
Return	LRESULT:	NIL		
Global Structure	Element		Receiving	Returning
	HWND			
	int			
	UINT FAR*			
	char FAR* FAR*			
	int FAR*			
	UINT FAR*			
	long FAR*			
	const char FAR*			

Received by:	MDM			
Name:	WM_INQUIRE		Value:	5560
Parameters	LPARAM:	NIL		
	WPARAM:	NIL		
Return	LRESULT:	NUMBER OF BP HELD BY MDM		
Global Structure	Element		Receiving	Returning
	HWND			
	int			
	UINT FAR*			
	char FAR* FAR*			
	int FAR*			
	UINT FAR*			
	long FAR*			
	const char FAR*			

Received by:	MDM			
Name:	WM_GET_FILE_NAME		Value:	5557
Parameters	LPARAM:	BUTTON NUMBER (MDM BODY PART SEQ NUMBER)		
	WPARAM:	NIL		
Return	LRESULT:	NIL		
Global Structure	Element	Receiving	Returning	
	HWND			
	int			
	UINT FAR*			
	char FAR* FAR*			
	int FAR*			
	UINT FAR*			
	long FAR*			
	const char FAR*		FILE NAME & PATH	

Received by:	X.400 APPLICATION			
Name:	WM_MDM_CLOSED		Value:	5558
Parameters	LPARAM:	NIL		
	WPARAM:	NIL		
Return	LRESULT:	NIL		
Global Structure	Element	Receiving	Returning	
	HWND			
	int			
	UINT FAR*			
	char FAR* FAR*			
	int FAR*			
	UINT FAR*			
	long FAR*			
	const char FAR*			

Received by:	X.400 APPLICATION			
Name:	WM_SEND_TO_MDM		Value:	5561
Parameters	LPARAM:	BP SEQUENCE NUMBER		
	WPARAM:	NIL		
Return	LRESULT:	BOOL TRUE		
Global Structure	Element		Receiving	Returning
	HWND			
	int			
	UINT FAR*			
	char FAR* FAR*			
	int FAR*			
	UINT FAR*			
	long FAR*			
	const char FAR*		FILE NAME TO BE CREATED	

Appendix 2 - C++ Listing for the MDM

MDMEX.H:

```
// application class
#include "resource.h"

#define     MAX_BUTTONS        28          // max number of dynamic buttons
#define          BUTTONSIZE         39      // size of the buttons (square)
#define          SPACE               5      // space between each button and the
window
#define          FILE_NAME_START "__bp_"
#define          NUMBER_BP_SENT      5          // number of body parts sent by
Register
#define          SIZE_TO_LOAD 300000// max size to load automatically = 300K


const       int movie = 1;
const       int sound = 2;
const       int image = 3;
const       int   text  = 4;
const       int unknown = 5;


// frame window class
class CMainFrame : public CFrameWnd
{
public:
    CMainFrame();

protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg void OnClose();
    afx_msg BOOL OnQueryEndSession();
    afx_msg void OnHlp();
      afx_msg void OnFileNew();
      afx_msg void OnFileAddbodypart();
      afx_msg BOOL OnLaunch(UINT nID);
      afx_msg void OnFileExit();
      afx_msg LRESULT OnCallerHwnd(WPARAM hCallerHwnd, LPARAM);
      afx_msg LRESULT OnRegisterMMBP(WPARAM, LPARAM);
      afx_msg LRESULT OnZoomitClosed(WPARAM, LPARAM);
      afx_msg LRESULT OnInquiry(WPARAM, LPARAM);
      afx_msg LRESULT OnGetFile(WPARAM, LPARAM);
    //}}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()

public:

      void Initialize();
      BOOL CheckAssociation(CString Type);
      void DecideButton(const char* Filename);
      void ShowButton(UINT Button_type, const char* FullPath);
      void LoadFile(int ButtonNumber);

// Variables

      BOOL CallerPresent;                        // flag to check if X.400 app is
there

      CString gTempDirectory;                    // temporary directory to store files

      CStringArray  gNamesToZoomit;         // file names passed to Zoomit

      CStringArray  gFileEx;                     // array of file extensions
      int giFileExNo;                            // no of the file extension

      UINT giCallNumber[MAX_BUTTONS];       // Int for the incoming seq numbers

      int giNumberOfParts;                       // no of BP handled by MDM


      CBitmapButton  *gButton_Array[MAX_BUTTONS]; // array of buttons
      int giX_Pos, giY_Pos;                      // pos of buttons
      int giId;                                  // buttons Id
      CRect  gRect;

      int   giButton_Cnt;                        // button count

    // info attached to each button

      struct BodyPartInfo{
            CString      gFileNameWithPath;
            long  giFileSize;
            BOOL  gbLoadedFlag;
            UINT  giSeqNumber;
      };

      struct BodyPartInfo gItem[28];                    // one for each button

      HWND hMdm;          // HWND for the MDM
      HWND hCaller;             // HWND for the X.400 app

      // globally allocated struct for passing parameters

      struct ParameterList
      {
            HWND WindowHandle;              // window handles
            int NumberOfBodyParts;          // size of the arrays, no of BP offered
            UINT far* SequenceNumbers;      // their sequence numbers
```

```
            char far* far* BodyPartTypes;  // their types
            int far* MDM_BodyPartsHandled; // where to put the no of BP handled by MDM
            UINT far* MDM_SequenceNumbers; // where to put the Seq No for the above
            long far* BodyPartSize;              // Size of the BP
            const char far* FileToBeCreated; // File Name
        };

        struct ParameterList FAR* lpStruct;

};

//    Application Class

class CMdmexApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
```

MDMEX.CPP:

```
#include "stdafx.h"
#include "mdmex.h"
#include "addbodyp.h"
#include "dllapi.h"

CMdmexApp NEAR theApp; // the one and only CMdmexApp object

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
//    Function:    CMdmExApp::InitInstance
//
//    Purpose:     This function was copied from another program.
//                       It performs the initialization of the instance
//                       of the application and creates the CMainFrame
//                       object.
//
//    Returns:     BOOL TRUE.
//    ------------------------------------------------------------------ **

BOOL CMdmexApp::InitInstance()
{
    m_pMainWnd = new CMainFrame();
    ASSERT(m_pMainWnd != NULL); // error checking only
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();

    return TRUE;
```

```
}

/////////////////////////////////////////////////////////////////////////////
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_CLOSE()
    ON_WM_QUERYENDSESSION()
    ON_COMMAND(ID_HLP, OnHlp)
      ON_COMMAND(ID_FILE_NEW, OnFileNew)
      ON_COMMAND(ID_FILE_ADDBODYPART, OnFileAddbodypart)
      ON_COMMAND(ID_FILE_EXIT, OnFileExit)
      ON_COMMAND_EX(BID1, OnLaunch)
      ON_COMMAND_EX(BID2, OnLaunch)
      ON_COMMAND_EX(BID3, OnLaunch)
      ON_COMMAND_EX(BID4, OnLaunch)
      ON_COMMAND_EX(BID5, OnLaunch)
      ON_COMMAND_EX(BID6, OnLaunch)
      ON_COMMAND_EX(BID7, OnLaunch)
      ON_COMMAND_EX(BID8, OnLaunch)
      ON_COMMAND_EX(BID9, OnLaunch)
      ON_COMMAND_EX(BID10, OnLaunch)
      ON_COMMAND_EX(BID11, OnLaunch)
      ON_COMMAND_EX(BID12, OnLaunch)
      ON_COMMAND_EX(BID13, OnLaunch)
      ON_COMMAND_EX(BID14, OnLaunch)
      ON_COMMAND_EX(BID15, OnLaunch)
      ON_COMMAND_EX(BID16, OnLaunch)
      ON_COMMAND_EX(BID17, OnLaunch)
      ON_COMMAND_EX(BID18, OnLaunch)
      ON_COMMAND_EX(BID19, OnLaunch)
      ON_COMMAND_EX(BID20, OnLaunch)
      ON_COMMAND_EX(BID21, OnLaunch)
      ON_COMMAND_EX(BID22, OnLaunch)
      ON_COMMAND_EX(BID23, OnLaunch)
      ON_COMMAND_EX(BID24, OnLaunch)
      ON_COMMAND_EX(BID25, OnLaunch)
      ON_COMMAND_EX(BID26, OnLaunch)
      ON_COMMAND_EX(BID27, OnLaunch)
      ON_COMMAND_EX(BID28, OnLaunch)
      ON_MESSAGE(WM_CALLER_HWND, OnCallerHwnd)
      ON_MESSAGE(WM_REGISTER, OnRegisterMMBP)
      ON_MESSAGE(WM_ZOOMIT_CLOSED, OnZoomitClosed)
      ON_MESSAGE(WM_INQUIRE, OnInquiry)
      ON_MESSAGE(WM_GET_FILE_NAME, OnGetFile)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
```

```
//     Function:    CMainFrame::CMainFrame (CMainFrame Constructor)
//
//     Purpose:     This function was copied from another program. It performs
//                        the construction of the CMainFrame object when invoked
//                        by the InitInstance function new operator.
//
//     Returns:     Nothing.
//     ------------------------------------------------------------------ **


CMainFrame::CMainFrame()
{
      CRect MdmExRect (5, 380, 635, 480);   // size and location of the window

    Create(NULL, "Multimedia Display Module",
          WS_OVERLAPPEDWINDOW, MdmExRect, NULL,
          MAKEINTRESOURCE(IDR_MAINFRAME)); // creation of the window

      giNumberOfParts = 0;                  // initialized here once. when first
                                                  // invoked, there are no body
parts
                                                  // handled by the MDM.
Initialize
                                                  // makes a decision based on
that
                                                  // value.

      Initialize();                              // make the proper initializations

      CallerPresent = FALSE;                // here because Initialize
                                                  // is called by functions
                                                  // called by Caller!!

    hMdm = GetSafeHwnd();            // Get the HWND to the MDM Window

}

//     ------------------------------------------------------------------ **
//     ** Multimedia Communications Research Laboratory
//     ** Department of Electrical Engineering, University of Ottawa
//
//     By:                Sylvain Carrier
//     Date:        24 March 1994
//
//     Function:    Initialize()
//
//     Purpose:     This function initializes all the variables that are
//                        involved in the generation of buttons.  It also
//                        retrieves the location of the temporary environment
//                        variable for the storage of temporary files.  If
//                        files were created as a result of a Register function,
//                        it removes them.  The decision is based on the number
//                        of body parts accepted by MDM.  If not 0,
//                        files were created and have to be deleted.  If there
//                        are not any, no files have to be deleted.
```

```
//
//    Returns:     nothing
//    ------------------------------------------------------------ **

void CMainFrame::Initialize()
{
      giY_Pos = SPACE;          // location of the first button Y-axis
      giX_Pos = SPACE;          // location of the first button X-axis
      giId = BID1;                    // Identification number of the first button

      int init_cnt;


//    initializing the button information array

      for (init_cnt=0; init_cnt!=28; init_cnt++)
      {
            gItem[init_cnt].gFileNameWithPath = "";
            gItem[init_cnt].giFileSize = 0;
            gItem[init_cnt].gbLoadedFlag = FALSE;
            gItem[init_cnt].giSeqNumber = 0;
      }

      giButton_Cnt = 0;   // initial button count = 0


//    routine to find the temporary directory on the user's machine
//    usually TEMP or TMP
//    if nothing found, will use root (\)

      gTempDirectory = getenv("TEMP");

      if(gTempDirectory.IsEmpty())
      {
            gTempDirectory = getenv("TMP");
      }

      if(gTempDirectory.IsEmpty())
      {
            gTempDirectory = getenv("temp");
      }

      if(gTempDirectory.IsEmpty())
      {
            gTempDirectory = getenv("tmp");
      }

      if(!(gTempDirectory.IsEmpty()))
      {
            gTempDirectory += "\\";
      }

      giFileExNo = 0;       // for the array of extension names
                                    // size is as many as body parts received
                                    // because have to create the array
```

```
                                        // before checkAssociation

//      here, files created from a previous Register command are erased, if any

        for(init_cnt=0; init_cnt < giNumberOfParts; init_cnt++) // if giNumberOfParts is
not 0,
        {

                remove(gNamesToZoomit[init_cnt]);                       // remove the files.
        }


        giNumberOfParts = 0;            // initialize to zero now that possible
                                                        // files where erased

}


//      ------------------------------------------------------------------ **
//      ** Multimedia Communications Research Laboratory
//      ** Department of Electrical Engineering, University of Ottawa
//
//      By:                     Sylvain Carrier
//      Date:           13 June 1994
//
//      Function:       OnRegisterMMBP
//
//      Purpose:        This function will delete existing buttons and
//                              initialize the variables (OnFileNew).  It reads
//                              the Multimedia body parts given by Zoomit, checks
//                              for an association and provides answers to Zoomit as
//                              to what body parts are accepted.  It then asks
//                              Zoomit to copy the body parts to filenames given
//                              and displays the buttons associated with the filenames
//                              passed to X.400 application. If the files are too big,
//                              the transfer will not be requested to X.400 application.
//
//      Returns:        Nothing.
//      ------------------------------------------------------------------ **


LRESULT CMainFrame::OnRegisterMMBP(WPARAM, LPARAM)
{

        CallerPresent = TRUE;     // if register was received, there is an
                                                // X.400 application calling!


        OnFileNew();                            // Remove existing buttons and Initialize

//      the X.400 application can send a Register with no body parts.
//      this is used to clear all buttons for a message that has
//      no multimedia body parts.
```

```
	if(lpStruct->NumberOfBodyParts == 0)                    // if no body parts
	{
		return NULL;                    // it was just to clear the buttons, return
	}

	int i;                                                  // Int for the For loop

	CStringArray BPType;                                // Local array for Body Part
Types

	BPType.SetSize(lpStruct->NumberOfBodyParts, -1);  // Size the array according to
the Number of BP
	gFileEx.SetSize(lpStruct->NumberOfBodyParts);      // Size the file ext array
according to what is presented
                                                                // required at
this time because needed in FindAssociation


//	this loop is where each body part type is verified against the
//	user's configuration to see if it can be displayed

	for (i=0; i < lpStruct->NumberOfBodyParts; i++)
	{
		BPType[i] = *lpStruct->BodyPartTypes++;            // get the body part type
		UINT SeqNumber = *lpStruct->SequenceNumbers++;     // get incoming sequence
number
		long FileSize = *lpStruct->BodyPartSize++;         // get incoming body part
size

	  if (CheckAssociation(BPType[i]))    // check if there is an association
		{                                                       // if so,
			giNumberOfParts += 1;                          // increment the
no of parts handled
			*lpStruct->MDM_SequenceNumbers++ = SeqNumber;      // give the
current seq number to Zoomit
		giCallNumber[giFileExNo++] = SeqNumber; // keep a copy of the sequence
number
		gItem[i].giSeqNumber = SeqNumber;       // attach the seqnumber to the
button

		gItem[i].giFileSize = FileSize;       // attach the file size to the button
		}
	}

	*lpStruct->MDM_BodyPartsHandled = giNumberOfParts;       // give the total no of
parts handled to Zoomit

	gNamesToZoomit.SetSize(giNumberOfParts);    // set the number of filenames
required
                                                                // setting
the size of CString Array

//	this loop is where file names are created for the body parts
//	to be received from the X.400 application
```

```
	char temp[4;;	// for the itoa function

	for(i=0; i != giNumberOfParts; i++)
	{
		gNamesToZoomit[i] = gTempDirectory;		// temp directory found earlier
		gNamesToZoomit[i] += FILE_NAME_START; // __bp_ is beginning file name
		gNamesToZocmit[i] += *itoa(i,temp,10);		// arbitrary seq number
		gNamesToZoomit[i] += ".";
		gNamesToZoomit[i] += gFileEx[i];		// file extension from type given

		DecideButton(gNamesToZoomit[i]);		// decide on the type and show
button
	}

	return 0;		// return value
has no meaning

}

//	---------------------------------------------------------------- **
//	** Multimedia Communications Research Laboratory
//	** Department of Electrical Engineering, University of Ottawa
//
//	By:		Sylvain Carrier
//	Date:		24 March 1994
//
//	Function:	CheckAssociation(CString Type)
//
//	Purpose:	This function checks the WIN.INI to see if there is
//			an executable associated with the Type given.
//
//	Returns:	True if there is an association for the Type given
//			False if there is no association for the Type given
//	---------------------------------------------------------------- **


BOOL CMainFrame::CheckAssociation(CString Type)
{

	Type.MakeUpper();  // ensures that the string is all in caps

//	we have to get rid of any characters over three

	if (Type == "JPEG")
	{
		Type = "JPG";
	}

	if (Type == "MPEG")
	{
		Type = "MPG";
	}

	int numberbytes = 0;
	PSTR pszBuf;
```

```
	pszBuf = (PSTR) LocalAlloc(LMEM_FIXED, 80);
	numberbytes = GetProfileString("Extensions", Type, "", pszBuf, 80);
	LocalFree((HANDLE) pszBuf);

//	if numberbytes is not zero, there is an association

	if (numberbytes != 0)
	{
		gFileEx[giFileExNo] = Type;
		return TRUE;
	}
	else
	{
		return FALSE;
	}

}

//	------------------------------------------------------------------ **
//	** Multimedia Communications Research Laboratory
//	** Department of Electrical Engineering, University of Ottawa
//
//	By:			Sylvain Carrier
//	Date:		24 March 1994
//
//	Function:	DecideButton(const char* Filename)
//
//	Purpose:	This function decides on the type of button that
//				is shown based on the extension extracted from
//				Filename and creates a button that will launch
//				the file given by Filename.
//
//	Returns:	Nothing.
//	------------------------------------------------------------------ **


void CMainFrame::DecideButton(const char* Filename)
{
	CString FileAndPath;
	FileAndPath = (CString)Filename;

	CString extension;
	extension = FileAndPath;			// get the whole name & path
	extension.MakeUpper();				// all upper case
	extension.MakeReverse();			// start from the end
	extension = extension.SpanExcluding(".");	// take everything until the "."
	extension.MakeReverse();			// put back in the proper order
	                                                // this is
required for extensions with
	                                                // less than
three characters (.AU)

//	if the file corresponds to an image file extension
```

```
      if (extension == "PCX" || extension == "GIF" || extension == "JPG" || extension
== "BMP")
      {
            ShowButton(image, FileAndPath);
            return;
      }

//    mpeg movie file extension

      if (extension == "MPG")
      {
            ShowButton(movie, FileAndPath);
            return;
      }

//    sound file extension

      if (extension == "AU" || extension == "WAV")
      {
            ShowButton(sound, FileAndPath);
            return;
      }

//    text file extension

      if (extension == "TXT" || extension == "WRI")
      {
            ShowButton(text, FileAndPath);
            return;
      }

//    if no type found, show a button with a question mark!

      ShowButton(unknown, FileAndPath);

}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        24 March 1994
//
//    Function:    ShowButton(UINT Button_type, const char* FullPath)
//
//    Purpose:     This function will show a button of type Button_type
//                       (movie, sound, image or text) that will launch the
//                       the file given at FullPath.
//
//    Returns:     nothing
//    ------------------------------------------------------------------ **


void CMainFrame::ShowButton(UINT Button_type, const char* FullPath)
```

```
{
//     the maximum number of button was arbitrarily set to 28
//     that corresponds to two rows at VGA resolution

       if (giButton_Cnt == 28) return;       // too many, do not display

       UINT Button_up;           // variables to hold resource Ids
       UINT Button_down;

     switch (Button_type)
       {
             case movie:

                   Button_up =   IDB_BITMAP_MOVIE_UP;
                   Button_down = IDB_BITMAP_MOVIE_DOWN;

                   break;

             case sound:

                   Button_up =    IDB_BITMAP_SOUND_UP;
                   Button_down = IDB_BITMAP_SOUND_DOWN;

                   break;

             case image:

                   Button_up =    IDB_BITMAP_IMAGE_UP;
                   Button_down = IDB_BITMAP_IMAGE_DOWN;

                   break;

             case text:

                   Button_up =   IDB_BITMAP_TEXT_UP;
                   Button_down = IDB_BITMAP_TEXT_DOWN;

                   break;

             default:

                   Button_up =   IDB_BITMAP_UNKNOWN_UP;
                   Button_down = IDB_BITMAP_UNKNOWN_DOWN;

       }

       gRect.top = giY_Pos;                  // size and location of the button
       gRect.left = giX_Pos;
       gRect.bottom = giY_Pos+BUTTONSIZE;
       gRect.right = giX_Pos+BUTTONSIZE;
       giX_Pos += BUTTONSIZE + SPACE;

//     if it is the 13th button, it is time to change row!
```

```
	if (giButton_Cnt == 13)
	{
		giY_Pos += BUTTONSIZE + SPACE;
		giX_Pos = SPACE;
	}

//	creation of the button routine

	gButton_Array[giButton_Cnt] = new CBitmapButton;
	gButton_Array[giButton_Cnt]->Create("BuT",
WS_CHILD|WS_VISIBLE|WS_TABSTOP|BS_PUSHBUTTON|BS_OWNERDRAW, gRect, this, giId++);
	gButton_Array[giButton_Cnt]->LoadBitmaps(MAKEINTRESOURCE(Button_up),
MAKEINTRESOURCE(Button_down));

	gItem[giButton_Cnt].gFileNameWithPath = FullPath; // assign file name to button

	LoadFile(giButton_Cnt);                                          // ask to
load the file

	giButton_Cnt = giButton_Cnt + 1;                                 // get ready for
next button

}


//	------------------------------------------------------------ **
//	** Multimedia Communications Research Laboratory
//	** Department of Electrical Engineering, University of Ottawa
//
//	By:		Sylvain Carrier
//	Date:		20 June 1994
//
//	Function:	CMainFrame::LoadFile
//
//	Purpose:	This function decides if there is a file to be loaded
//			by checking the size of the corresponding file. If the
//			size is not NULL, then a message is sent to the calling
//			application if and only if the size of the file is not
//			over a predetermined value (set at 300K for the prototype).
//
//	Returns:	Nothing.
//	------------------------------------------------------------ **

void CMainFrame::LoadFile(int ButtonNumber)
{

//	if the size is zero, there is nothing to be loaded
//	this is used as a flag to indicate that the file is local
//	and does not require to be loaded

	if(gItem[ButtonNumber].giFileSize == 0)
	{
		return;			// nothing to load or local file
	}
```

```
//    if it is small enough, ask the X.400 app to send it in!

      if(gItem[ButtonNumber].giFileSize < SIZE_TO_LOAD)
      {
            lpStruct->FileToBeCreated = gItem[ButtonNumber].gFileNameWithPath;
                  // put the file name that has to be created for X.400 app
            LRESULT Loaded = ::SendMessage(hCaller, WM_SEND_TO_MDM,
gItem[ButtonNumber].giSeqNumber, 0L);

            // if it was loaded, change the corresponding flag so it is known that
            // the file was loaded.

            if (Loaded)
            {
                  gItem[ButtonNumber].gbLoadedFlag = TRUE;
            }
      }

}


//                      M E S S A G E     H A N D L E R S


//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:               Sylvain Carrier
//    Date:       20 June 1994
//
//    Function:   CMainFrame::OnClose [Message Handler]
//
//    Purpose:    This function reacts to the WM_CLOSE message and
//                      shows a message box to confirm the desire to
//                      close the app.
//
//    Returns:    Nothing.
//    ------------------------------------------------------------------ **

void CMainFrame::OnClose()
{
    if (AfxMessageBox("OK to close window?", MB_YESNO) == IDYES)
    {
        OnFileExit();  //    after confirmation, close properly
    }
}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:               Sylvain Carrier
//    Date:       20 June 1994
```

```
//
//     Function:    CMainFrame::OnQueryEndSession [Message Handler]
//
//     Purpose:     This function responds to the WM_QUERYENDSESSION
//                        message and sends a WM_CLOSE message.
//
//     Returns:     TRUE
//     ------------------------------------------------------------------ **

BOOL CMainFrame::OnQueryEndSession()
{
    // received when user quits Windows
    SendMessage(WM_CLOSE);
    return TRUE;
}


//     ------------------------------------------------------------------ **
//     ** Multimedia Communications Research Laboratory
//     ** Department of Electrical Engineering, University of Ottawa
//
//     By:                Sylvain Carrier
//     Date:         20 June 1994
//
//     Function:    CMainFrame::OnHlp [Message Handler]
//
//     Purpose:     This function is called when the user hits the Help
//                        menu item. The function calls the dialog prepared
//                        in the AppStudio to show app information.
//
//     Returns:     Nothing.
//     ------------------------------------------------------------------ **

void CMainFrame::OnHlp()
{

    CDialog(IDD_DIALOG1).DoModal();

}

//     ------------------------------------------------------------------ **
//     ** Multimedia Communications Research Laboratory
//     ** Department of Electrical Engineering, University of Ottawa
//
//     By:                Sylvain Carrier
//     Date:         24 March 1994
//
//     Function:    OnFileNew() [Message Handler]
//
//     Purpose:     This function will delete all existing buttons and
//                        call the Initialize() function.
//
//     Returns:     Nothing.
//     ------------------------------------------------------------------ **
```

```
void CMainFrame::OnFileNew()
{
      // delete buttons

      for(giButton_Cnt; giButton_Cnt != 0; giButton_Cnt--)
      {
            delete gButton_Array[giButton_Cnt-1];
      }

      // reinitialize

      Initialize();

}

//    ------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:               Sylvain Carrier
//    Date:       24 March 1994
//
//    Function:   OnOpenBodyPart() [Message Handler]
//
//    Purpose:    This function allows the user to select a filename
//                      when clicking the File Add Body Part menu option.
//                      The file selected will be attached to a button and
//                      a button will be shown.
//
//    Returns:    Nothing.
//    ------------------------------------------------------------ **


void CMainFrame::OnFileAddbodypart()
{

//    if there is no X.400 application present, this function will not
//    perform anything.

    if(!CallerPresent)
    {
      ::MessageBox(hMdm, "There is no Client to receive the Body Parts",
                                    "No Client Application", MB_ICONSTOP | MB_OK);

      return;
    }

    CAddBodyPart dlg;          // declares dlg a CAddBodyPart object

//    invoke the dialog to load a file

      if (dlg.DoModal()== IDCANCEL)
      {
            return;
```

```
      }

      CString pFileName_ext = dlg.GetPathName();  // get the file name with path

      DecideButton(pFileName_ext);                        // show the corresponding
button
}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:               Sylvain Carrier
//    Date:       24 March 1994
//
//    Function:   OnLaunch(UINT nID) [Message Handler]
//
//    Purpose:    This function will launch the file associated with
//                      the button that was clicked as identified by nID.
//
//    Returns:    Always returns true.
//;   ------------------------------------------------------------------ **


BOOL CMainFrame::OnLaunch(UINT nID)
{

      //    if the size of the file is not zero, it may have to be loaded

    if (!gItem[nID-BID1].giFileSize == 0)   // if not 0, then load may be required
    {
            int Answer = 0;

            //    if the loaded flag is not set, than the file is not loaded
            //    the user will be asked if a loading is desired

            if(!gItem[nID-BID1].gbLoadedFlag)
            {
                  Answer = ::MessageBox(hMdm, "Hit Ok to load and present the Body
Part\n                Hit Cancel to skip",
                                    "Body Part Not Loaded", MB_ICONEXCLAMATION |
MB_OKCANCEL);
            }


      if (Answer == IDCANCEL)
      {
            return FALSE;      //    not loaded
      }

      if (Answer == IDOK)
      {
                  lpStruct->FileToBeCreated = gItem[nID-BID1].gFileNameWithPath;
                        // put the file name that has to be created
                        // and ask the X.400 app to send it in!
```

```
                    LRESULT Loaded = ::SendMessage(hCaller, WM_SEND_TO_MDM, gItem[nID-
BID1].giSeqNumber, 0L);

                    // if it was loaded, then the corresponding flag

                    if (Loaded)
                    {
                          gItem[nID-BID1].gbLoadedFlag = TRUE;
                    }

      }
    }       // if the file size was 0, it is a local file than can be displayed

    HCURSOR curs_saved = SetCursor(LoadCursor(NULL, IDC_WAIT));      //     show the
hourglass

    // start the viewer with the filename as the argument

      ShellExecute(m_hWnd,NULL,gItem[nID-BID1].gFileNameWithPath,NULL,NULL,SW_SHOW);
      SetCursor(curs_saved);

      return TRUE;
}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:               Sylvain Carrier
//    Date:       20 June 1994
//
//    Function:   OnFileExit() [Message Handler]
//
//    Purpose:    This function reacts to the File Exit menu item.
//                      It deletes the buttons, removes files left by
//                      a Register, informs the X.400 app that MDM is closing
//                      and closes MDM.
//
//    Returns:    Nothing.
//    ------------------------------------------------------------------ **

void CMainFrame::OnFileExit()
{

      for(giButton_Cnt; giButton_Cnt != 0; giButton_Cnt--)
      {
            delete gButton_Array[giButton_Cnt-1];
      }

      int i;
      for(i=0; i < giNumberOfParts; i++)    // if giNumberOfParts is not 0,
      {

            remove(gNamesToZoomit[i]);                    // remove the files.
```

```
        }

        if(CallerPresent)
        {
                ::SendMessage(hCaller, WM_MDM_CLOSED, 0, 0L);
        }

        CFrameWnd::OnClose();
}

//      ------------------------------------------------------------------ **
//      ** Multimedia Communications Research Laboratory
//      ** Department of Electrical Engineering, University of Ottawa
//
//      By:                 Sylvain Carrier
//      Date:           20 June 1994
//
//      Function:       OnCallerHwnd [Message Handler]
//
//      Purpose:        This function is called when a WM_CALLER_HWND message
//                              is received. It reads the X.400 app HWND and the pointer
//                              the the struct that is used to exchange parameters.
//
//      Returns:        The HWND for the MDM is returned in the WindowHandle
//                              element of the struct.
//      ------------------------------------------------------------------ **

LRESULT CMainFrame::OnCallerHwnd(WPARAM hCallerWnd, LPARAM lpTemp)
{

        hCaller = (HWND)hCallerWnd;
        lpStruct = (struct ParameterList far *)lpTemp;

        lpStruct->WindowHandle = hMdm;

        return 0;
}

//      ------------------------------------------------------------------ **
//      ** Multimedia Communications Research Laboratory
//      ** Department of Electrical Engineering, University of Ottawa
//
//      By:                 Sylvain Carrier
//      Date:           20 June 1994
//
//      Function:       OnZoomitClosed [Message Handler]
//
//      Purpose:        This function is called when a WM_ZOOMIT_CLOSED message
//                              is received. It sets the CallerPresent flag to False
//                              and returns.
//
//      Returns:        Nothing.
//      ------------------------------------------------------------------ **

LPESULT CMainFrame::OnZoomitClosed(WPARAM, LPARAM)
```

```
{
      CallerPresent = FALSE;
      return NULL;
}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
//    Function:    OnInquiry [Message Handler]
//
//    Purpose:     This function is called when a WM_INQUIRE message is
//                       received. It returns the count of buttons displayed
//                       in the MDM. Note that the count is set to the next
//                       button (not there yet).
//
//    Returns:     Actual Button Count.
//    ------------------------------------------------------------------ **

LRESULT CMainFrame::OnInquiry(WPARAM, LPARAM)
{
      return giButton_Cnt;
}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
//    Function:    OnGetFile [Message Handler]
//
//    Purpose:     This function is called when a WM_GET_FILE_NAME message
//                       is received. It reads a button number (body part
//                       sequence number for the MDM).
//
//    Returns:     The file name and path for the button number received
//                       is returned in the FileToBeCreated element of the struct.
//    ------------------------------------------------------------------ **

LRESULT CMainFrame::OnGetFile(WPARAM Number, LPARAM)
{
      lpStruct->FileToBeCreated = gItem[Number].gFileNameWithPath;
      return NULL;
}
```

Appendix 3 - C++ Listing for the X.400 Emulator

CALLER.H:

```
// application class
#include "resource.h"

#define            NUMBER_BP_SENT      6          // number of body parts sent by
Register

class CCallerApp : public CWinApp
{
public:
    virtual BOOL InitInstance();

};


// frame window class
class CMainFrame : public CFrameWnd
{
public:
    CMainFrame();
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg void OnClose();
    afx_msg BOOL OnQueryEndSession();
    afx_msg void OnHlp();
      afx_msg void OnRegister();
      afx_msg void OnFileExit();
      afx_msg void OnInquire();
      afx_msg void OnGetfile();
      //}}AFX_MSG
    DECLARE_MESSAGE_MAP()

private:

      LRESULT OnMdmClosed(WPARAM, LPARAM);
      LRESULT OnSendFileToMDM(WPARAM, LPARAM);


    void InitializeRegisterData();

    void PrepareStruct();
```

```
      int copyfile(char *source, char *target);

      void NotifyMdm();


//    Structure that holds data to MDM

      struct ParameterList
      {
            HWND WindowHandle;
            int NumberOfBodyParts;          // size of the arrays
            UINT far* SequenceNumbers;
            char far* far* BodyPartTypes;
            int far* MDM_BodyPartsHandled;
            UINT far* MDM_SequenceNumbers;
            long far* BodyPartSize;
            const char far* FileToBeCreated;
      };

      HGLOBAL hgStruct;
      struct ParameterList FAR* lpStruct;


//    Register body part info
//    public for the caller
//    passed to MDM

      CString  Source[NUMBER_BP_SENT];

      int NoBP;
      UINT SeqNum[NUMBER_BP_SENT];
      char far* Types[NUMBER_BP_SENT];
      int MDMNoBpHandled;
      UINT MDM_SeqNum[NUMBER_BP_SENT];
      long BodyPartSize[NUMBER_BP_SENT];

      int far* pMDMNoBpHandled;

      BOOL MdmExWindowCreated;
      HWND hCaller;
      HWND hMdm;
      UINT MdmExInstance;

      LRESULT InquiryResult;

      int ScreenX, ScreenY;

};
```

CALLER.CPP:

```
#include "stdafx.h"
#include "caller.h"
```

```
//  for the copy file function begin
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <fcntl.h>          /* _O_ constant definitions */
#include <sys\types.h>
#include <sys\stat.h>       /* _S_ constant definitions */
#include <malloc.h>
#include <errno.h>
//    for the copy file function end


CCallerApp NEAR theApp; // the one and only CCallerApp object

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
//    Function:    CCallerApp::InitInstance
//
//    Purpose:     This function was copied from another program.
//                       It performs the initialization of the instance
//                       of the application and creates the CMainFrame
//                       object.

//    Returns:     BOOL TRUE.
//    ------------------------------------------------------------------ **

BOOL CCallerApp::InitInstance()
{
    m_pMainWnd = new CMainFrame();
    ASSERT(m_pMainWnd != NULL); // error checking only
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}


/////////////////////////////////////////////////////////////////////////////
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_CLOSE()
    ON_WM_QUERYENDSESSION()
    ON_COMMAND(ID_HLP, OnHlp)
      ON_COMMAND(ID_REGISTER, OnRegister)
      ON_COMMAND(ID_FILE_EXIT, OnFileExit)
      ON_COMMAND(ID_INQUIRE, OnInquire)
      ON_MESSAGE(WM_MDM_CLOSED, OnMdmClosed)
      ON_COMMAND(ID_GETFILE, OnGetfile)
      ON_MESSAGE(WM_SEND_TO_MDM, OnSendFileToMDM)
      //}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////
//    ---------------------------------------------------------- **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:         20 June 1994
//
//    Function:     CMainFrame::CMainFrame (CMainFrame Constructor)
//
//    Purpose:      This function was copied from another program. It performs
//                        the construction of the CMainFrame object when invoked
//                        by the InitInstance function new operator.
//
//    Returns:      Nothing.
//    ---------------------------------------------------------- **

CMainFrame::CMainFrame()
{
    Create(NULL, "Caller Application",
           WS_OVERLAPPEDWINDOW, rectDefault, NULL,
           MAKEINTRESOURCE(IDR_MAINFRAME));

    hCaller = GetSafeHwnd();    // get HWND for the Caller

    PrepareStruct();            // allocate struct in global memory

    InitializeRegisterData();   // initialize the data required
                                        // to test the MDM

    MdmExWindowCreated = FALSE;       // set the flag to indicate that there
                                            // is no MDM yet.

}

//    ---------------------------------------------------------- **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:         24 March 1994
//
//    Function:     InitializeRegisterData()
//
//    Purpose:      This function initializes the data required to
//                        emulate the Register function call that will be
//                        done by the X.400 app.
//
//    Returns:      nothing
//    ---------------------------------------------------------- **


void CMainFrame::InitializeRegisterData()
```

```
{
        NoBP = 6;                       // number of body parts offered to MDM
        SeqNum[0] = 11;                 // sequence numbers (don't have to be
        SeqNum[1] = 2;                  // in sequence or anything specific
        SeqNum[2] = 34;
        SeqNum[3] = 4;
        SeqNum[4] = 56;
        SeqNum[5] = 666;
        Types[0] = "au";          // types of the BP

        //    file names for the BP

        Source[0] = "c:\\fichiers\\programs\\junk\\bark.AU";
        Types[1] = "MpEG";
        Source[1] = "c:\\fichiers\\programs\\junk\\xtitle.mpg";
        Types[2] = "jpeg";
        Source[2] = "c:\\fichiers\\programs\\junk\\bigcat.jpg";
        Types[3] = "waV";
        Source[3] = "c:\\fichiers\\programs\\junk\\chimes.wav";
        Types[4] = "tXt";
        Source[4] = "c:\\fichiers\\programs\\junk\\PaPer.TxT";
        Types[5] = "AU";
        Source[5] = "c:\\fichiers\\programs\\junk\\monkey.au";

        MDMNoBpHandled = 0;             // initialize the BP handled by MDM
        MDM_SeqNum[0] = 0;        // initialize the seq numbers for the above
        MDM_SeqNum[1] = 0;
        MDM_SeqNum[2] = 0;
        MDM_SeqNum[3] = 0;
        MDM_SeqNum[4] = 0;
        MDM_SeqNum[5] = 0;

        BodyPartSize[0] = 59L;          // put in fictitious sizes
        BodyPartSize[1] = 1000L;
        BodyPartSize[2] = 10000L;
        BodyPartSize[3] = 50000L;
        BodyPartSize[4] = 310000L;      // this is above the size limit for imm. load
        BodyPartSize[5] = 100000L;



        pMDMNoBpHandled = &MDMNoBpHandled;    // address passed on to MDM

        InquiryResult = 0;        // initial result of inquiry to MDM
        ScreenX = 5;              // initial settings for caller screen
        ScreenY = 0;

}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
```

```
//     Date:        20 June 1994
//
//     Function:    CMainFrame::PrepareStruct
//
//     Purpose:     This function allocates global heap memory for the
//                        struct that is used to pass the parameters between
//                        the two apps.
//
//     Returns:     Nothing.
//     ------------------------------------------------------------------ **

void CMainFrame::PrepareStruct()
{

      hgStruct = GlobalAlloc(GMEM_SHARE, sizeof(struct ParameterList));
      lpStruct = (struct ParameterList FAR *)GlobalLock(hgStruct);

}

//     ------------------------------------------------------------------ **
//     ** Multimedia Communications Research Laboratory
//     ** Department of Electrical Engineering, University of Ottawa
//
//     By:                Sylvain Carrier (taken from COPY1.C in C/C++ Help)
//     Date:        24 March 1994
//
//     Function:    copyfile(char *source, char *target)
//
//     Purpose:     This function is required to perform the Zoomit
//                        emulation.  It copies the file given by source to the
//                        name and location given by target.
//
//     Returns:     an integer that represents the error no.  It is
//                        not read nor used.
//     ------------------------------------------------------------------ **


int CMainFrame::copyfile(char *source, char *target)
{

/* Copies one file to another (both specified by path). Dynamically
 * allocates memory for the file buffer. Prompts before overwriting
 * existing file. Returns 0 if successful, otherwise an error number.
 */

      char *buf;
    int hsource, htarget;
    unsigned count = 0xff00;

    // Open source file and create target, overwriting if necessary. */
    if( (hsource = _open( source, _O_BINARY | _O_RDONLY )) == - 1 )
        return errno;
    htarget = _open( target, _O_BINARY | _O_WRONLY | _O_CREAT | _O_TRUNC,
                             _S_IREAD | _S_IWRITE );
```

```
    if( htarget == -1 )
        return errno;

    if( (unsigned)_filelength( hsource ) < count )
        count = (int)_filelength( hsource );

    /* Dynamically allocate a large file buffer. If there's not enough
     * memory for it, find the largest amount available on the near heap
     * and allocate that. This can't fail, no matter what the memory model.
     */
    if( (buf = (char *)malloc( (size_t)count )) == NULL )
    {
        count = _memmax();
        if( (buf = (char *)malloc( (size_t)count )) == NULL )
            return ENOMEM;
    }

    /* Read-write until there's nothing left. */
    while( !_eof( hsource ) )
    {
        /* Read and write input. */
        if( (count = _read( hsource, buf, count )) == -1 )
            return errno;
        if( (count = _write( htarget, buf, count )) == - 1 )
            return errno;
    }

    /* Close files and release memory. */
    _close( hsource );
    _close( htarget );
    free( buf );
    return 0;

}


//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
//    Function:    CMainFrame::NotifyMdm
//
//    Purpose:     This function will notify the MDM that the X.400
//                       application has been closed.
//
//    Returns:     Nothing.
//    ------------------------------------------------------------------ **

void CMainFrame::NotifyMdm()
{
      if(MdmExWindowCreated)
      {
```

```
            ::SendMessage(hMdm, WM_ZOOMIT_CLOSED, 0, 0L);
        }

        GlobalUnlock(hgStruct);              // global memory cleanup
        GlobalFree(hgStruct);
}


/////////////////////////////////////////////////////////////////////////////
//                  M E S S A G E   H A N D L E R S
/////////////////////////////////////////////////////////////////////////////
//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                 Sylvain Carrier
//    Date:         20 June 1994
//
//    Function:     CMainFrame::OnClose [Message Handler]
//
//    Purpose:      This function is called when a WM_CLOSE message is
//                        received. It confirms that the user wishes to exit,
//                        notifies the MDM and closes the app.
//
//    Returns:      Nothing.
//    ------------------------------------------------------------------ **

void CMainFrame::OnClose()
{
    if (AfxMessageBox("OK to close window?", MB_YESNO) == IDYES)
    {
      NotifyMdm();
        CFrameWnd::OnClose();
    }
}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                 Sylvain Carrier
//    Date:         20 June 1994
//
//    Function:     CMainFrame::OnQueryEndSession [Message Handler]
//
//    Purpose:      This function is called when a WM_ENDQUERYSESSION
//                        message is received. It then sends the WM_CLOSE message
//                        to confirm with the user and properly close the app.
//
//    Returns:      Nothing.
//    ------------------------------------------------------------------ **

BOOL CMainFrame::OnQueryEndSession()
{
    // received when user quits Windows
```

```
    SendMessage(WM_CLOSE);
    return TRUE;
}

////////////////////////////////////////////////////////////////////////
//    ---------------------------------------------------------------- **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
//    Function:    CMainFrame::OnHelp [Message Handler]
//
//    Purpose:     This function is called when the user hits Help.
//                       This function does nothing!
//
//    Returns:     Nothing.
//    ---------------------------------------------------------------- **

void CMainFrame::OnHlp()
{
//    TRACE("Entering CMainFrame::OnHlp\n");
}

//    ---------------------------------------------------------------- **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
//    Function:    CMainFrame::OnRegister [Command Handler]
//
//    Purpose:     This function is called when the user hits Register.
//                       It launches the MDM, gets the MDM HWND and concludes
//                       with a WM_REGISTER message to emulate multimedia BP
//                       sent to the MDM.
//
//    Returns:     Nothing.
//    ---------------------------------------------------------------- **

void CMainFrame::OnRegister()
{
    //launch an MDM only if is the first time register is called

      if (!MdmExWindowCreated)
      {
            MdmExInstance = WinExec("c:\\fichiers\\programs\\mdm\\mdmex.exe",
                                                      SW_SHOW);
            MdmExWindowCreated = TRUE;                  // set flag to indicate MDM is
there

            lpStruct->WindowHandle = hCaller;     // provide HWND to MDM
```

```
        //   Broadcast a message to all windows to obtain the MDM's HWND

      ::SendMessage(HWND_BROADCAST, WM_CALLER_HWND, (UINT)hCaller, (long)lpStruct);

      hMdm = lpStruct->WindowHandle;        // MDM returns his HWND in WindowHandle

      }

    CString TempFileName;

    //load the struct
    lpStruct->WindowHandle = NULL;
    lpStruct->NumberOfBodyParts = NoBP;
    lpStruct->SequenceNumbers = SeqNum;
    lpStruct->BodyPartTypes = Types;
    lpStruct->MDM_BodyPartsHandled = pMDMNoBpHandled;
    lpStruct->MDM_SequenceNumbers = MDM_SeqNum;
    lpStruct->BodyPartSize = BodyPartSize;
    lpStruct->FileToBeCreated = TempFileName;

    LRESULT Reply = ::SendMessage(hMdm, WM_REGISTER, 0, 0L);

}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
//    Function:    CMainFrame::OnFileExit [Command Handler]
//
//    Purpose:     This function is called when the user hits File Exit.
//                       It notifies the MDm and closes without confirmation
//                       with the user.
//
//    Returns:     Nothing.
//    ------------------------------------------------------------------ **

void CMainFrame::OnFileExit()
{

    NotifyMdm();
      CFrameWnd::OnClose();

}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:        20 June 1994
//
```

```
//     Function:    CMainFrame::OnMdmClosed [Message Handler]
//
//     Purpose:     This function is called when a WM_MDM_CLOSED message
//                        is received. It resets the flag that indicates
//                        MDM's presence.
//
//     Returns:     Nothing.
//     ------------------------------------------------------------------ **

LRESULT CMainFrame::OnMdmClosed(WPARAM, LPARAM)
{

      MdmExWindowCreated = FALSE;

      return NULL;
}

//     ------------------------------------------------------------------ **
//     ** Multimedia Communications Research Laboratory
//     ** Department of Electrical Engineering, University of Ottawa
//
//     By:                Sylvain Carrier
//     Date:        20 June 1994
//
//     Function:    CMainFrame::OnInquire [Command Handler]
//
//     Purpose:     This function is called when the Inquiry menu item
//                        is selected. It inquires with the MDM how many body
//                        parts are available to be sent with a message.
//
//     Returns:     Nothing.
//     ------------------------------------------------------------------ **

void CMainFrame::OnInquire()
{
      // if there is no MDM, return!
      if(!MdmExWindowCreated)
      {
            return;
      }

      // send the message to MDM
      InquiryResult = ::SendMessage(hMdm, WM_INQUIRE, 0, 0L);

      // Display the result in the caller window
      CClientDC dc(this);
      char buffer[5];
      itoa((int)InquiryResult, buffer, 10);
      CString temp = "MDM has the following number of body parts: ";
      temp += buffer;
      dc.TextOut(ScreenX, ScreenY, temp);
      ScreenY += LINEADVANCE;

}
```

```
//     ------------------------------------------------------------ **
//     ** Multimedia Communications Research Laboratory
//     ** Department of Electrical Engineering, University of Ottawa
//
//     By:                Sylvain Carrier
//     Date:         20 June 1994
//
//     Function:     CMainFrame::OnGetfile [Message Handler]
//
//     Purpose:      This function is called when the Get File Name command
//                         is selected. It checks if a MDM is present, sends an
//                         Inquiry message to check if there are BP for the X.400
//                         app and then sends the WM_GET_FILE_NAME message to
//                         obtain the file names from the MDM.
//
//     Returns:      Nothing.
//     ------------------------------------------------------------ **

void CMainFrame::OnGetfile()
{

      // if no MDM, return!
      if(!MdmExWindowCreated)
      {
            return;
      }

      InquiryResult = ::SendMessage(hMdm, WM_INQUIRE, 0, 0L);

      // if no BP, return!
      if(InquiryResult == 0)
      {
            return;
      }

      CClientDC dc(this);

      // prepare a CString array with the result of the inquiry
      int i;
      CStringArray FileNames;
      FileNames.SetSize((int)InquiryResult, -1);

      //    for InquiryResult -1, send the get file name message
      //    and display on the screen
      for(i=0; i<InquiryResult; i++)
      {
            ::SendMessage(hMdm, WM_GET_FILE_NAME, i, 0L);
            FileNames[i] = (char far*)lpStruct->FileToBeCreated;

      dc.TextOut(ScreenX, ScreenY, FileNames[i]);
      ScreenY += LINEADVANCE;

      }
```

```
}

//    ------------------------------------------------------------------ **
//    ** Multimedia Communications Research Laboratory
//    ** Department of Electrical Engineering, University of Ottawa
//
//    By:                Sylvain Carrier
//    Date:         20 June 1994
//
//    Function:     CMainFrame::OnSendFileToMDM [Message Handler]
//
//    Purpose:      This function is called when a WM_SEND_TO_MDM message
//                        is received. It compares the sequence number given by
//                        MDM with those in the array of seq numbers and when
//                        a match is found, the corresponding file is copied
//                        to the file name and location provided by MDM.
//
//    Returns:      BOOL TRUE.
//    ------------------------------------------------------------------ **

LRESULT CMainFrame::OnSendFileToMDM(WPARAM SequenceNumber, LPARAM)
{
    char psource[50], pdest[50];

    int j;
    for(j=0; j < NoBP; j++)
    {
      //    find the corresponding sequence number
      if(SeqNum[j] == SequenceNumber)
      {
            // get the corresponding file name in psource
            strcpy (psource, Source[j]);
            break;
      }
    }

    //get the file name and location desired
    CString temp = (char far*)lpStruct->FileToBeCreated;

    //put it in pdest
      strcpy (pdest, temp);

      //    and copy the file
      copyfile(psource, pdest);

      return TRUE;
}
```