Messaging Service Access Modules

This chapter describes Apple Open Collaboration Environment (AOCE) messaging service access modules. A messaging service access module is a software component that provides the PowerTalk user with access to external mail and messaging services. You do not need to read this chapter if you are writing a mail or messaging application or adding mail or messaging capabilities to your application.

To write a messaging service access module, you need to be familiar with many components of AOCE software. You should read the chapters "Introduction to the Apple Open Collaboration Environment" and "AOCE Utilities" in *Inside Macintosh: AOCE Application Interfaces* before reading this chapter to get a general overview of AOCE software components and the shared AOCE data types and the utility routines that act on them. This chapter assumes that you are familiar with AOCE catalogs and records and their structures, and that you know how to read and write data to them. The chapters "Standard Catalog Package" and "Catalog Manager" in *Inside Macintosh: AOCE Application Interfaces* describe the high-level application programming interface (API) and the low-level API to AOCE catalogs, respectively.

To read and write AOCE records, you must obtain an authentication identity. Identities are described in the chapter "Authentication Manager" in *Inside Macintosh: AOCE Application Interfaces.*

Along with your messaging service access module, you need to provide a type of AOCE template called an *address template* to allow the user to enter address information. If you are writing a personal messaging service access module, you also need to provide a setup template that allows the user to configure your access module. The chapter "AOCE Templates" in *Inside Macintosh: AOCE Application Interfaces* describes how to write an AOCE template. The chapter "Service Access Module Setup" in this book provides additional specific information about setup and address templates and their interaction with messaging service access modules and the PowerTalk Key Chain.

All messaging service access module developers need to be familiar with high-level events. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information about high-level events.

This chapter starts with an introduction to messaging service access modules. Subsequent sections describe

- personal messaging service access modules
- server messaging service access modules
- the types of messages that are read and written by messaging service access modules
- AOCE addresses
- the AOCE high-level events
- how to get messages out of an AOCE system
- how to put messages into an AOCE system
- the structures and routines in the messaging service access module API

Introduction to Messaging Service Access Modules

A ***messaging system*** is a combination of hardware and software that provides people and processes with the ability to exchange electronic messages—it provides messaging services. Apple's ***AOCE messaging system*** consists of PowerTalk system software and PowerShare mail servers that allow Macintosh users and processes accessible over a network or via a modem to exchange electronic messages. Today there are many types of messaging systems, such as Internet, AppleLink, QuickMail, and so forth, with which AOCE users might want to communicate. To facilitate the exchange of messages between an AOCE messaging system and other existing and future messaging systems, the AOCE architecture defines a ***messaging service access module (MSAM).*** An MSAM links Apple's AOCE messaging system to another messaging system, extending the reach of messaging service clients.

The AOCE architecture defines two kinds of MSAMs. A ***personal MSAM*** translates messages and transfers them between a user's Macintosh and the user's account on another messaging system. It runs on a user's Macintosh. A ***server MSAM*** translates and transfers messages between a PowerShare mail server and a non-AOCE messaging system. A server MSAM transfers messages for any number of users located on the AppleTalk network to which it is connected. It runs on a Macintosh with a PowerShare mail server. Thus, the MSAM component of AOCE software architecture is scalable. It can provide service to a single user who uses a non-networked Macintosh computer or to large numbers of users in large internetworks.

Figure 2-1 shows how adding an MSAM to an AOCE system extends the reach of AOCE users. Prior to adding an MSAM, AOCE users cannot exchange electronic messages with others who are accessible only on a non-AOCE messaging system. Once an MSAM that connects to the non-AOCE messaging system is added, the AOCE users can exchange messages with people accessible on the non-AOCE messaging system.

The basic services provided by both personal and server MSAMs include

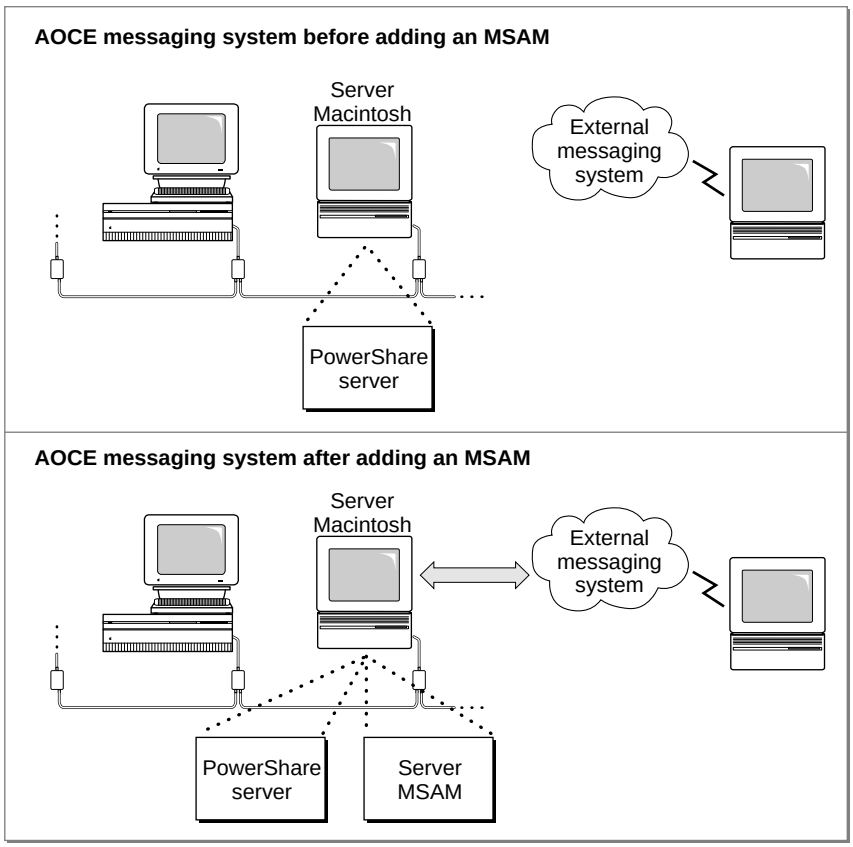
- transferring messages between an AOCE messaging system and another messaging system
- translating the content of messages between AOCE-defined formats and other formats
- translating message addresses between AOCE-defined formats and other formats
- reporting the results of attempts to deliver messages

Personal and server MSAMs are described in more detail in the following sections.

A note on terminology

Throughout this chapter, the term *message* is used as an inclusive term to refer to all types of messages. When information applies only to letters (a specific type of message), the term *letter* is used. When information applies only to messages that are not letters, the term ***non-letter message*** is used. Letters and messages are defined in the section "Types of Messages" beginning on page 2-16.

Figure 2-1 Adding an MSAM



Messaging systems that are not provided automatically with PowerTalk system software and PowerShare servers are collectively referred to as ***external messaging systems.*** An external messaging system may handle only letters or non-letter messages or both.

The term ***mail*** refers to letters. Messaging systems that handle only letters are sometimes referred to as *mail systems*.

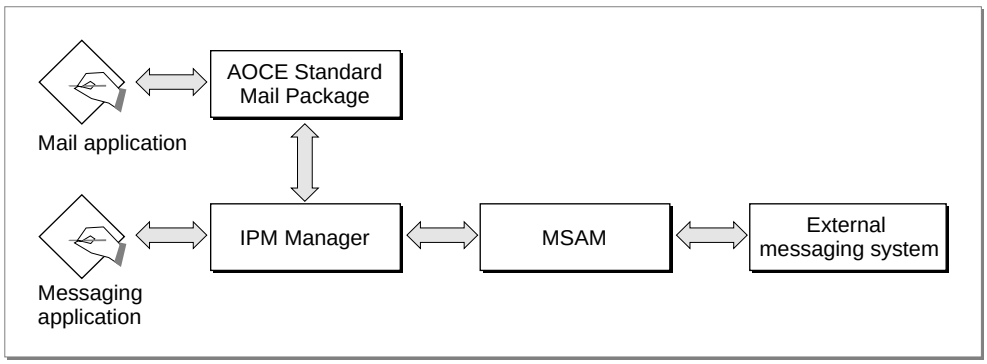
As a convention, this chapter refers to messages coming into an AOCE system from an external messaging system as ***incoming messages*** and to those that are leaving an AOCE system to go into an external messaging system as ***outgoing messages.***

Throughout the chapter, the text distinguishes between personal and server MSAMs where appropriate. The term *MSAM* is used when the text applies to both personal and server MSAMs, unless it is clear from the context that only a personal or server MSAM is meant. ◆

An MSAM is a low-level component in the AOCE software hierarchy. It does not directly provide services to a user or process; rather, it provides services indirectly through either the Standard Mail Package or the Interprogram Messaging (IPM) Manager. Thus, a client has a standard interface to all messaging systems, including those that are accessible via

MSAMs as well as Apple's PowerTalk and PowerShare services, regardless of underlying differences in how messages are accessed and formatted. Figure 2-2 shows the relationship of two clients, the Standard Mail Package, the IPM Manager, an MSAM, and an external messaging system.

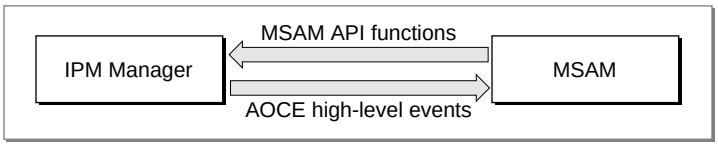
Figure 2-2 An MSAM's relationship to AOCE software



MSAMs interact with the IPM Manager. Either the MSAM or the IPM Manager can initiate communication with the other. Figure 2-3 illustrates the way the IPM Manager and an MSAM initiate communications with each other. An MSAM initiates communication with the IPM Manager by calling one of the functions provided in the MSAM API. These functions are described in detail in the section "MSAM Functions" beginning on page 2-130.

The IPM Manager initiates communication with an MSAM by sending it a high-level event. The events that the IPM Manager may send to an MSAM, which typically instruct the MSAM to take some action or advise it of a status change, are described in the section "High-Level Events" beginning on page 2-220.

Figure 2-3 Communication between the IPM Manager and an MSAM



Personal MSAMs

A personal MSAM allows a user or a mail or messaging application to transfer messages between the user's Macintosh and users or applications on one or more external messaging systems. A personal MSAM connects to an external messaging system and transfers messages between the user's Macintosh and the external messaging system. The user or process must have an account on the external messaging system to which the personal MSAM provides access. The user's Macintosh does not need to be connected to an AppleTalk network.

A personal MSAM is a background-only application; that is, it has no user interface.

Every personal MSAM must be accompanied by AOCE templates that allow the user to configure the MSAM and to enter address information. These templates, called the *setup template* and *address template*, are described in the chapter "Service Access Module Setup" in this book. Information that applies to all AOCE templates is provided in the chapter "AOCE Templates" in *Inside Macintosh: AOCE Application Interfaces*.

A file containing a personal MSAM must have a file type of either `'msam'` or `'csam'`. If you provide both a personal MSAM and a catalog service access module (CSAM) in the same file, use the file type `'csam'` (for "combined service access module"). If you provide a personal MSAM only, use the file type `'msam'`. You must include your setup and address templates in the same file as your personal MSAM.

Although personal MSAMs and server MSAMs both connect to external messaging systems and translate and transfer messages, there are a number of differences between them. See Table 2-1 on page 2-11 for a list of these differences.

A ***slot***, as the term is used in the MSAM API and in this chapter, refers to a collection of information about one account on an external messaging system. The information includes whatever is necessary to allow an MSAM to access the account and retrieve and send messages. Slot information determines what external messaging system the MSAM connects to. The term ***mail slot*** refers to a slot that allows the transfer of letters. The term ***messaging slot*** refers to a slot that allows the transfer of non-letter messages.

Slot information is stored in the form of AOCE record attributes in records in the PowerTalk Setup catalog. The record types in which the information is stored differ depending on whether you provide a combined MSAM/CSAM or a stand-alone MSAM. If you provide a combined MSAM/CSAM, slot information and its associated catalog information is stored in a single Combined record. If you provide a stand-alone MSAM, slot information is stored in a Mail Service record (sometimes called a *slot record*) and associated catalog information is stored in a Catalog record. The setup template that you provide with your MSAM writes slot information to some of these records; the PowerTalk Key Chain writes to others. The chapter "Service Access Module Setup" in this book describes the required attributes of the Combined, Mail Service, and Catalog records, and it explains who is responsible for writing those attributes to the different types of records in the Setup catalog.

In addition to the required record attributes, slot information includes whatever is necessary to allow the MSAM to service the slot—for instance, an access telephone number and the line speed. MSAMs can define record attribute types to store slot configuration information.

A personal MSAM can manage more than one slot. For example, if a user had two accounts on an external messaging system of a given type, a personal MSAM would manage two slots, one for each of the user's accounts on that messaging system. A personal MSAM also can connect to more than one external messaging system. For example, if a user has an account on each of two independent messaging systems, the same personal MSAM can connect to each system and manage a slot for the user's account there.

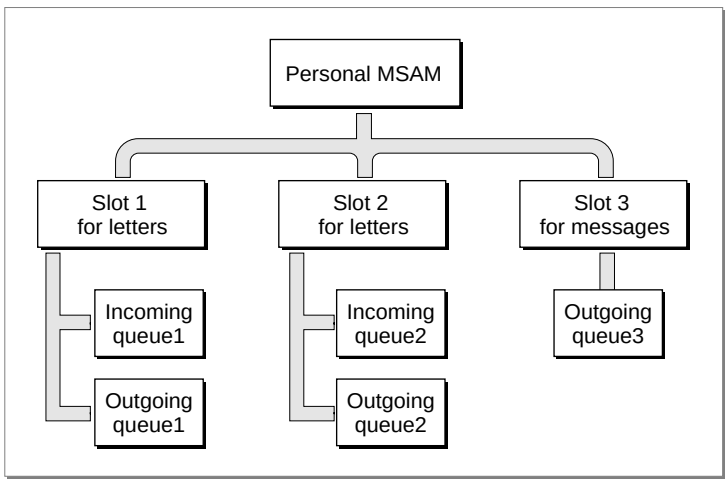
Each mail slot that a personal MSAM manages has two queues: an incoming queue and an outgoing queue.

Each messaging slot that a personal MSAM manages has an outgoing queue. The notion of an incoming queue does not apply to messaging.

An ***incoming queue*** contains AOCE letters that the personal MSAM translates from mail received from its external messaging system and each letter's associated message summary. (See the section "MSAM Modes of Operation" beginning on page 2-12 for information about message summaries.) An ***outgoing queue*** contains messages that the personal MSAM must deliver to an external messaging system. A personal MSAM retrieves a message from an outgoing queue, translates it, and delivers it to the intended recipients on the external messaging system.

Note that any given queue contains either letters and message summaries or non-letter messages. It does not contain both. Figure 2-4 shows an example of a personal MSAM with three slots and their associated queues.

Figure 2-4 Personal MSAM with its slots and queues



IMPORTANT
In release 1 of the AOCE software, the handling of non-letter messages is not fully supported for personal MSAMs. Therefore it is not advisable for a personal MSAM to implement the transfer of non-letter messages using release 1 of the AOCE software. ▲

Server MSAMs

A server MSAM allows users and processes on an AppleTalk network to exchange messages with other users and processes on one or more external messaging systems. It serves its clients indirectly by acting as a conduit for messages between a PowerShare mail server and the external systems to which the MSAM is connected. It must run on the same Macintosh as its PowerShare mail server.

Server MSAMs route messages between different messaging systems rather than between individual accounts on those systems. Therefore, a server MSAM does not necessarily need to know about specific accounts on an external messaging system, and, as a result, it has no concept of slots.

A server MSAM can connect to different types of messaging systems. For instance, a single server MSAM might connect to one or more Simple Mail Transfer Protocol (SMTP), X.400, and X.500 systems.

A server MSAM is a foreground Macintosh application. Once a server MSAM is launched, it should run continuously.

(A server MSAM and its PowerShare mail server do not have to run on a dedicated Macintosh. However, performance of other applications on the same Macintosh may suffer when the MSAM and server are very busy.)

Table 2-1 summarizes the differences between personal MSAMs and server MSAMs. (Not all of the differences have been discussed at this point.) You may want to refer to this table as you read succeeding sections in this chapter.

Table 2-1 Differences between personal MSAMs and server MSAMs

Characteristic	Personal MSAM	Server MSAM
Application type	Background-only	Foreground
Interconnects	User/process to specific account	Multiple users/processes to messaging system
Needs specific account information	Yes	No
Uses slots	Yes	No

continued

Table 2-1 Differences between personal MSAMs and server MSAMs (continued)

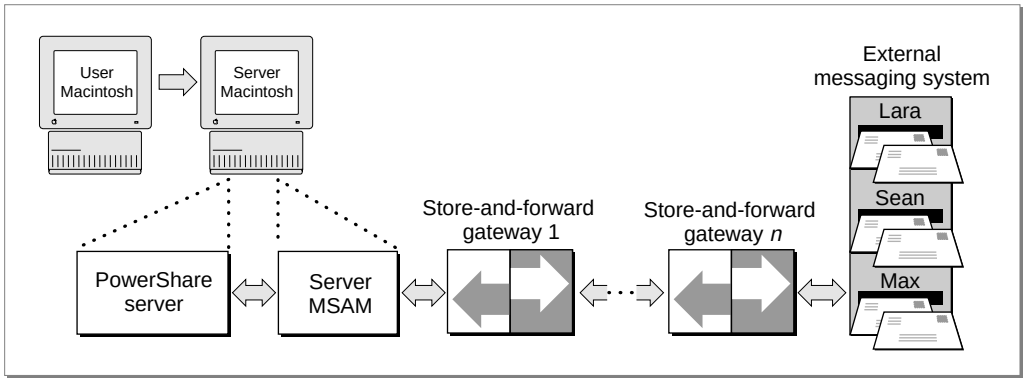
Characteristic	Personal MSAM	Server MSAM
Queues	1 outgoing queue per slot; 1 incoming queue per mail slot	1 outgoing queue
Writes message summaries	Yes	No
Can write incoming letters on demand	Yes	No
Needs setup template	Yes	No
Needs address template	Yes	Yes
Runs on	A user's Macintosh	A server Macintosh with a PowerShare mail server
Must be connected to an AppleTalk network	No	Yes
Transfers messages for more than one user	No	Yes
Mode of operation	Standard, online, quasi-batch	Standard
File type	`'csam'` or `'msam'`	`'APPL'`
Linked to its catalogs through	Mail Service and Catalog records in the Setup catalog	Foreign dNodes in AOCE catalog
Represented by	MSAM record in the Setup catalog	Forwarder record

MSAM Modes of Operation

In addition to its type (either personal or server), another important characteristic of an MSAM is its mode of operation. *Mode of operation* refers to the degree of control an MSAM retains over messages that it puts into an AOCE system. Some MSAMs function in some respects like a standard store-and-forward gateway; others function as an agent for the user. This section explains these modes in more detail.

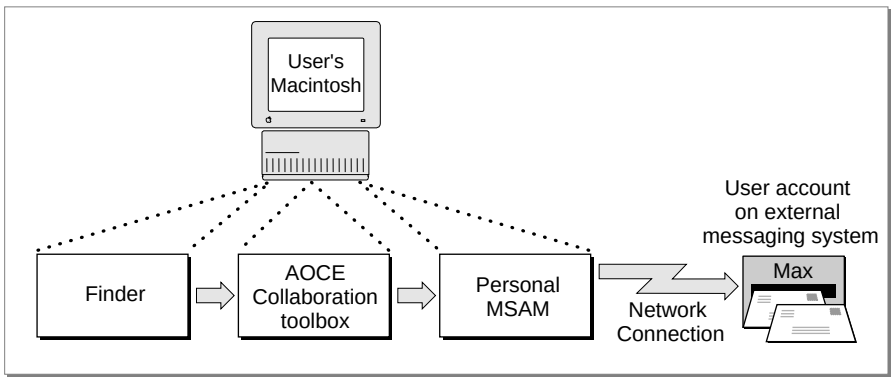
The store-and-forward gateway model consists of a source messaging system, a series of one or more store-and-forward gateways, and a destination messaging system. A ***store-and-forward gateway*** links different systems, providing temporary data storage and, where necessary, address translation. Figure 2-5 illustrates the store-and-forward gateway model. In such a model, the gateway hands off a message to the next link in the store-and-forward chain. Once it transfers a message, its responsibility for (and control of) that message ends. MSAMs that operate in this fashion are said to operate in ***standard mode.***

Figure 2-5 Store-and-forward gateway model



The online model consists of a source messaging system, a destination messaging system, and a personal MSAM that acts as an agent for the user in connecting those systems. In the online model, a personal MSAM does not act simply as a link in a series of store-and-forward gateways. Rather, it actively manages letters in a user's AOCE mailbox and in the user's accounts on external messaging systems, reflecting changes in one to the other, and keeping both ends synchronized as much as possible. Figure 2-6 illustrates the online model. MSAMs that operate in this fashion are said to operate in ***online mode.*** A personal MSAM operating in online mode can affect the user's experience quite directly, something an MSAM operating in standard mode cannot do.

Figure 2-6 Online model



A significant difference between standard mode and online mode is the point at which the MSAM is active. In standard mode, an MSAM is removed from any contact with the user. In online mode, the MSAM is actively involved with the user experience through the MSAM API and Finder interface.

A server MSAM always operates in standard mode. It delivers messages to a PowerShare mail server, at which point the MSAM's responsibility for the message ends. The AOCE system is responsible for delivering the message to its final destination. Similarly, from an AOCE system perspective, a server MSAM is a store-and-forward gateway in that messages sent to a server MSAM are addressed to a particular messaging system, not a specific address within that system.

A personal MSAM may operate in standard mode, online mode, or a variation of online mode referred to as *quasi-batch mode*. A personal MSAM always operates in standard mode when it is dealing with incoming non-letter messages. Much as a server MSAM hands off a message to a PowerShare mail server, the personal MSAM hands off a non-letter message to the IPM Manager resident on the Macintosh. Once it submits such a message to an AOCE system, the personal MSAM has no further control of or responsibility for the message. The AOCE system delivers the message to its final destination on the Macintosh. When a personal MSAM is dealing with incoming letters, however, it operates in online mode or quasi-batch mode.

IMPORTANT

A single personal MSAM may operate in both standard and online or quasi-batch modes; that is, it may handle both letters and non-letter messages. The MSAM API is general enough to cover all variations. As a result, the API contains features that do not apply in every case.

However, as noted earlier, the handling of non-letter messages is not fully supported for personal MSAMs in release 1 of the AOCE software. Therefore it is not advisable for a personal MSAM to implement the transfer of non-letter messages using release 1 of AOCE software. ▲

The AOCE software architecture allows a personal MSAM to operate in online mode (act as a user agent) by providing it with the means to deliver an incoming letter to a specific queue and to manipulate that letter after placing it in the queue.

The user's AOCE mailbox is a repository for letters from all of the different sources to which the user has access. These sources include an incoming queue for each mail slot managed by a personal MSAM installed on the Macintosh. On any given Macintosh with AOCE software installed, there are some number of destination queues for incoming messages, each of which contains either letters or non-letter messages. An incoming queue is a special type of destination queue for letters. It is special because a personal MSAM can manipulate an incoming queue and its contents. All other destination queues are under the control of the IPM Manager.

A personal MSAM submitting letters to an AOCE system must conform to certain minimal requirements of online mode. These requirements are to create, manage, and delete information blocks about the letters that it puts into an incoming queue. The information blocks are called ***message summaries.*** The AOCE Mailbox extension to the Finder uses message summaries to display information about the letters to the user. Message summaries are also the means by which a personal MSAM reflects changes in the status of a letter from the local Macintosh computer to the remote system and vice versa. Only personal MSAMs create message summaries for incoming letters.

Before a personal MSAM puts a letter into an incoming queue, it must first create the letter's message summary and put it into the incoming queue. A message summary contains

- information that is needed to display the letter to the user (this includes the subject of the letter, its timestamp, the sender's name, and so forth)
- status information, such as whether the user has read the letter or deleted the letter (a personal MSAM uses the status flags to maintain consistency between the letter's status on an AOCE system and on an external system)
- state information about the letter, such as whether the letter itself currently exists in the incoming queue
- whatever private data that you wish to attach to this letter (for instance, you may want to store the ID or reference number that uniquely identifies the letter on the external messaging system)

A message summary is defined by the `MSAMMsgSummary` structure, described on page 2-127.

After creating and submitting a message summary for a letter, a personal MSAM may immediately translate the letter into the AOCE letter format and put it into the incoming queue. Alternately, the MSAM can delay writing the letter until the user actually opens it. (The MSAM receives a high-level event when a user opens a letter.)

In general, a personal MSAM that connects to an external messaging system over a slow link should create the message summary and put the letter into the incoming queue at the same time. This gives a user faster access to the letter when he or she decides to read the letter. Also, when a link is slow or expensive, the MSAM might keep the copy of the letter the user has already read to avoid a retransmission if the user wants to read the letter again.

A personal MSAM that connects to an external messaging system over a fast link such as a local area network may choose to create just the message summary without automatically translating and transferring the letter itself. The MSAM can retrieve the letter on demand, that is, only when the user actually wants to read the letter. In these circumstances, it can delete the letter after the user reads it because retransmission would not cause much of a delay.

A personal MSAM may implement some features of online mode but not all, and it may thus operate somewhere in between standard and online modes. ***Quasi-batch mode*** represents a continuous gradation between standard and online modes. In quasi-batch mode, a personal MSAM may simply create a message summary, transfer the letter to an AOCE system, and do nothing further with regard to the letter. For example, a personal MSAM for fax transmissions might simply download a fax and put it into the incoming queue. Such a personal MSAM complies with only the minimal requirements of online mode and operates as much as possible like a standard store-and-forward gateway.

Table 2-2 shows the types of operating modes available to server and personal MSAMs.

Table 2-2 MSAM operating modes

Operating mode	Type of MSAM
Standard	Personal MSAM (for non-letter messages) and server MSAM
Online	Personal MSAM (for letters)
Quasi-batch	Personal MSAM (for letters)

This section has described the incoming queue as a special queue for incoming letters, available only to personal MSAMs with mail slots. There is no analogous construct on the outgoing side. All MSAMs, personal and server alike, have an outgoing queue from which they obtain outgoing messages. A server MSAM has a single outgoing queue that contains all of the messages addressed to external messaging systems to which it is connected. A personal MSAM, regardless of its operating mode, has one outgoing queue for each of its slots. Each queue contains the outgoing messages for the associated slot.

Types of Messages

The following sections discuss messages, letters, and reports.

Basic Messages

A ***message*** is the basic unit of communication defined by the Interprogram Messaging (IPM) Manager. A message consists of a message header followed by zero or more ***message blocks,*** each of which is a sequence of any number of bytes. The ***message header*** contains control information about the message, such as the message creator and message type, the total length of the message, the time it was submitted, addressing information, and so forth. It also contains the length, creator, and type of each block in the message. For more detailed information on the structure of messages and more information on the IPM Manager and the services it provides, see the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces*.

Every message has a message creator and a message type. The message creator and type are analogous to a Macintosh file's creator and type. The ***message creator*** indicates which application created the message. A ***message type*** indicates the semantics of the message, the type of blocks the message should contain, and the relationships among the various blocks in the message.

Similarly, every block has a block creator and a block type. The ***block creator*** indicates which application created the block. A ***block type*** indicates the format of the data contained within the block.

In addition to message types, AOCE software defines the concept of ***message families.*** A message that belongs to a message family shares a similar form with all other messages that belong to the same message family. Messages of the same family conform to the syntax of a defined set of message block types and their associated semantics. The syntax specifies which block types are optional and which are mandatory and specifies the relationships between the various blocks. Messages that belong to the same message family may also contain additional blocks whose types are not defined as part of the message family.

Apple defines three message families for an MSAM's use. All non-letter messages that an MSAM transfers belong to the `kIPMFamilyUnspecified` family. Letters may belong to either the `kMailFamily` or `kMailFamilyFile` family, both of which are defined in the next section. Although it is possible to distinguish a new class of messages by defining a new message family, it is not recommended that you do so.

IMPORTANT

Apple Computer, Inc., reserves all values for message and block types, message and block creators, and message families that consist entirely of lowercase letters and special characters. You are free to create and use other values except 0 and `'????'`. Apple Computer, Inc., does not provide a registry for message and block types, message and block creators, and message families. ▲

A message can contain another message. A message that is contained within another message is called a ***nested message.***

Letters

A ***letter*** is a type of message, consisting of a defined set of message blocks, that is intended to be read by a person.

A letter must contain a ***letter header block.*** A letter header block contains the address of the sender and of each recipient. It also contains the letter's attributes.

Letter attributes are bits of information about a letter. They include such things as the time the letter was sent, the subject of the letter, the priority assigned to the letter by the sender, and so forth.

Note

In this chapter, letter attributes are usually referred to simply as *attributes*. Do not confuse these letter attributes with record attributes. A record attribute refers to a part of an AOCE record. For information about record attributes, see the chapters "AOCE Utilities" and "Catalog Manager" in *Inside Macintosh: AOCE Application Interfaces.* ◆

A letter may have blocks that contain letter content, a nested letter, enclosures, and an image of the letter content. The MSAM API provides functions that you can use to read and write most of these blocks without specifying the block type. For example, the function `MSAMPutContent` automatically creates a block of type `kMailContentType`. However, to add a block of type image (`kMailImageBodyType`) or a private data block

(kMailMSAMType), you need to provide the block type to the MSAMPutBlock function. Table 2-3 lists the AOCE-defined block types that a letter may contain and the functions you use to read and write a block of a given type.

Table 2-3 Predefined letter block types

Block type	Value	Block contents	To read/write
kMailLtrHdrType	'lthd'	Letter header	MSAMGetRecipients MSAMPutRecipient MSAMGetAttributes MSAMPutAttribute
kMailContentType	'body'	Body of letter	MSAMGetContent MSAMPutContent
kMailEnclosureListType	'elst'	List of enclosures	MSAMGetEnclosure MSAMPutEnclosure
kMailEnclosureDesktopType	'edsk'	Desktop Manager information for enclosures	MSAMGetEnclosure MSAMPutEnclosure
kMailEnclosureFileType	'asgl'	A file enclosure	MSAMGetEnclosure MSAMPutEnclosure
kMailImageBodyType	'imag'	Image of letter	MSAMGetBlock MSAMPutBlock
kMailMSAMType	'gwyi'	MSAM-defined information	MSAMGetBlock MSAMPutBlock
kIPMEnclosedMsgType	'emsg'	Nested letter	MSAMOpenNested MSAMBeginNested
kIPMDigitalSignature	'dsig'	Digital signature	MSAMGetBlock MSAMPutBlock

Letter content is that part of the letter that the sender typically wants the recipient to read first, like the body of a conventional hard-copy letter. Letter content may be in three forms:

- a content block (block type is kMailContentType)
- an image block (block type is kMailImageBodyType)
- a content enclosure (block type is kMailEnclosureFileType)

A ***content block*** contains the body of a letter in one or more data segments. Each segment contains data of one of the following types:

- Plain text. A text segment contains data in one or more character sets (Roman, Arabic, Kanji, and so on) with 1-byte or 2-byte character codes, depending on the character set.

- Styled text. The segment contains text and a `StScrpRec` structure containing the style information for that text.
- Pictures. The segment contains data in PICT format.
- Sounds. The segment contains data in Audio Interchange File Format (AIFF).
- Movies. The segment contains data in QuickTime movie file format ('`MooV`').

These five data formats are collectively called *standard content* or ***standard interchange format,*** sometimes referred to as *AppleMail format*. All MSAMs must support standard content to facilitate interoperability. Any user with AOCE software installed can read and write letters containing standard content using the AppleMail application.

Another way of communicating a letter's content is to include it in an ***image block.*** Data in an image block is stored in a structure of type `TPfPgDir` followed by picture elements (PICTs). The format of data in an image block is sometimes referred to as *snapshot format*.

The AppleMail application can read image blocks. Thus, by including an image block in a letter, an application that uses formats other than standard interchange format can ensure that a user having the AppleMail application can view the formatted content. A receiver cannot edit image data. MSAMs should support image blocks.

The third form in which letter content may be transmitted or received is a ***content enclosure,*** sometimes referred to as a *main enclosure*. Such an enclosure is typically in the native format of the sending application. An MSAM is not required to support translations of various application file formats. A recipient must have a copy of the sending application to read a content enclosure. A letter can have only one content enclosure.

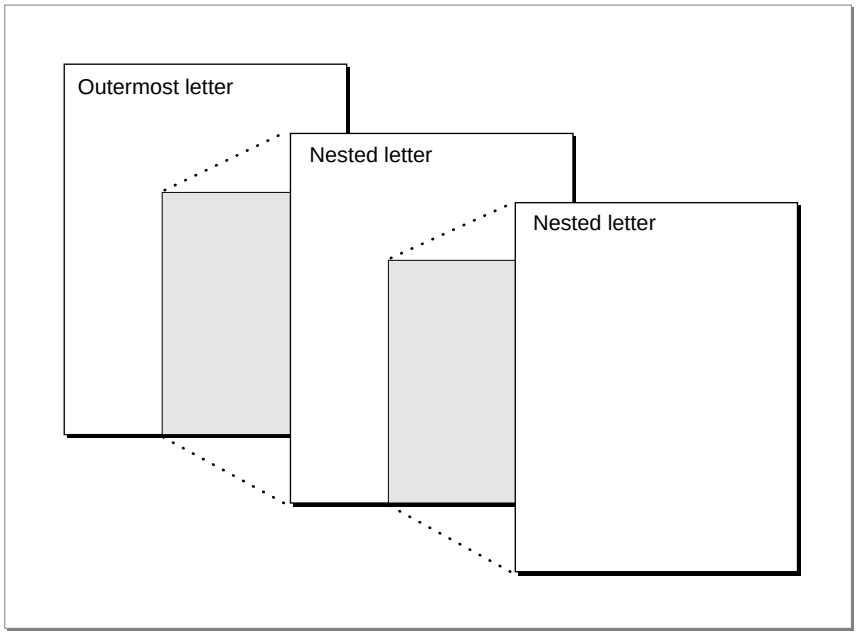
The contents (if any) of a letter may be in any or all of these three forms. Typically, you can expect letters to contain a content block as well as a content enclosure.

An ***enclosure*** is a file or folder sent along with a letter. An enclosure may be either a regular enclosure or a content enclosure. A ***regular enclosure*** is a file or folder included in a letter like an attachment in a conventional hard-copy letter. That letter may or may not contain a content block.

A letter can have up to 50 enclosures. An enclosure file can be of any type. If an enclosure is a folder, it can contain any number of files of any type, so long as the total number of enclosures does not exceed 50. Each file and folder counts as one enclosure. For example, if a letter had as an enclosure a folder containing three files, the total number of enclosures in the letter is four: one folder and three files. A content enclosure counts when totaling the number of enclosures in a letter.

A ***nested letter*** is a complete letter included whole within another letter. A letter can have only one letter nested within it. However, the nested letter itself may contain a nested letter. Figure 2-7 illustrates this concept.

Figure 2-7 Nested letters



The ***nesting level*** of a letter indicates how many letters are nested within it. The nesting level of a letter that contains no nested letters is 0. A letter that contains a letter with a nesting level of n has a nesting level of $n + 1$. Thus, if a reply letter contains a copy of the original letter, the nesting level of the reply is one greater than the nesting level of the original letter. Figure 2-8 illustrates an example of nesting letters. Sue sends a memo to Dan. Her original memo has a nesting level of 0. Dan replies to Sue and includes a copy of Sue's original memo in the reply. His reply has a nesting level of 1. Sue sends a different memo to Tim and includes Dan's reply. The nesting level of her memo to Tim is 2. The theoretical limit to the number of nesting levels is very large.

A forwarded letter is always a nested letter. It is nested within a letter that has no content and no enclosures. The letter that contains the forwarded letter has a nesting level of $n + 1$, where n is the nesting level of the forwarded letter.

Figure 2-8 How the nesting level increments

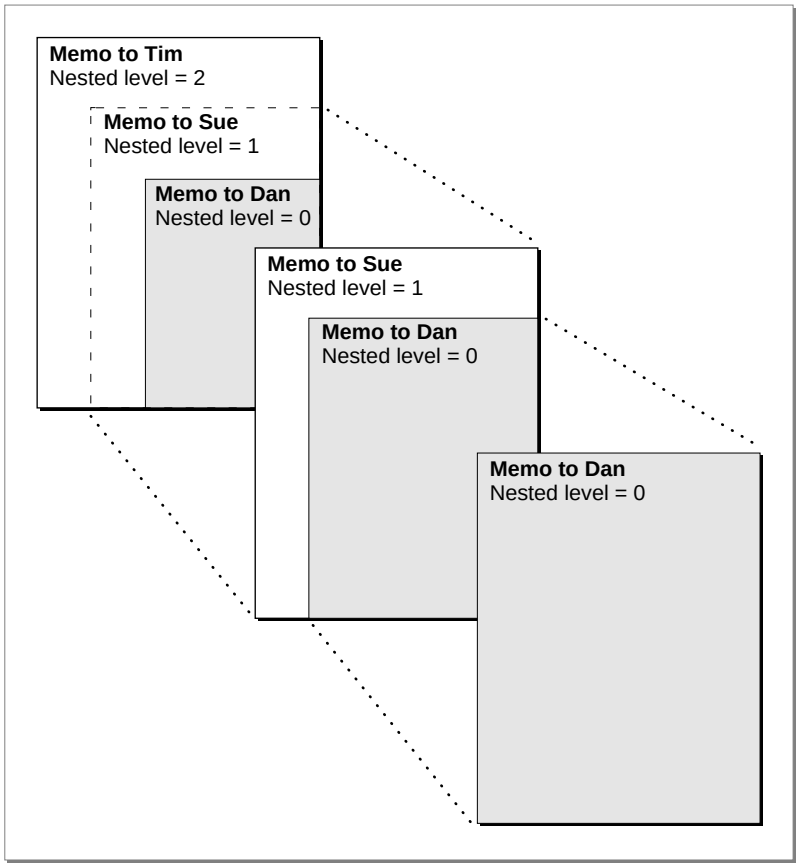
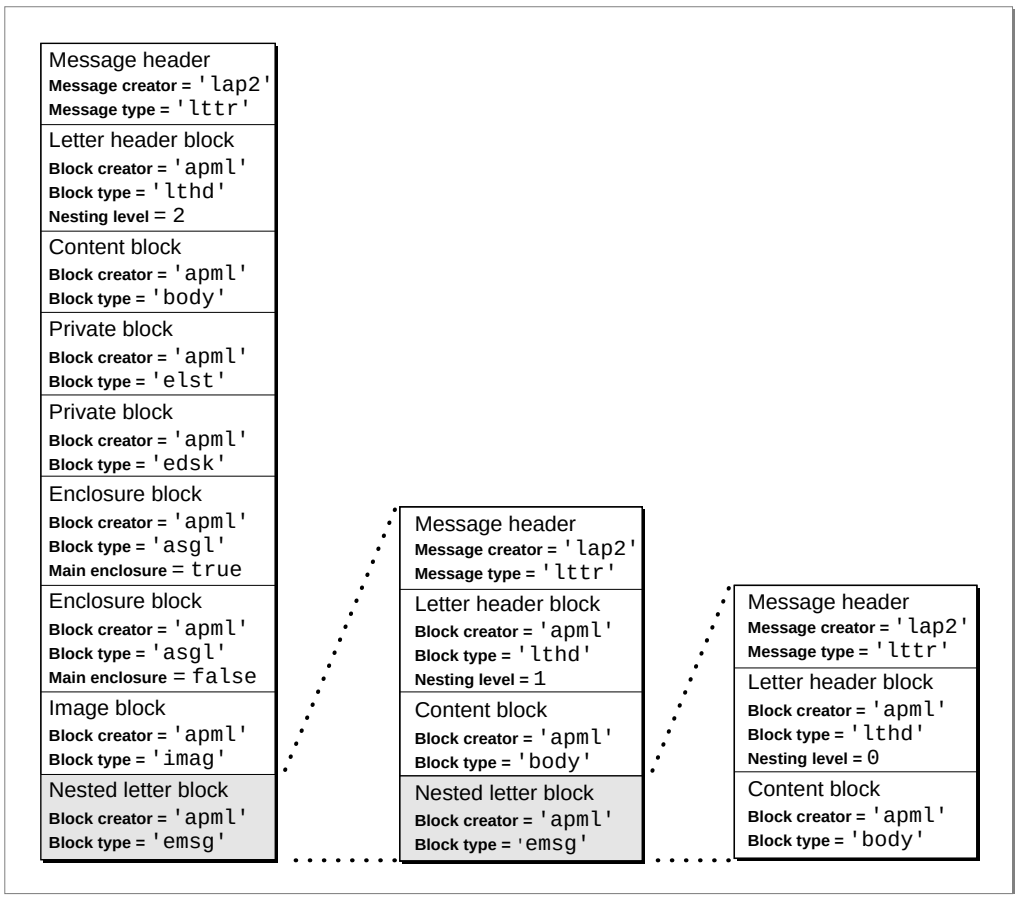


Figure 2-9 illustrates the structure of a hypothetical letter. In the message header, the message creator and type (`'lap2'` and `'lttr'`) indicate that this message is a letter that was created by the AppleMail application. Next is the letter header block. The letter header information includes the letter's nesting level, set to 2, indicating that this letter has two letters nested within it. The letter contains a content block. The blocks of type `kMailEnclosureListType` (`'elst'`) and `kMailEnclosureDesktopType` (`'edsk'`) are private to Macintosh system software. There are two enclosures in the letter, one of which is a content enclosure. An image block is present. It contains an alternate representation of the data in the content block. The letter also contains a nested letter in a

nested letter block. The nested letter is a complete letter consisting of a message header, a letter header block, a content block, and a nested letter block. Its letter header shows that its nesting level is 1. The nested letter block contains a complete letter consisting of a message header, a letter header block, and a content block. Its nesting level is 0.

Figure 2-9 Structure of a letter



Ordinarily, letters belong to one of two message families defined by AOCE software. A letter that belongs to the `kMailFamily` family may contain either a content block or any type of enclosure or both. A letter that belongs to the `kMailFamilyFile` family does not contain a content block or a content enclosure, but may contain a regular enclosure. You should not put a content block into or expect to get a content block from a letter in the `kMailFamilyFile` family.

Reports

A ***report*** communicates delivery information about a message to the sender of the message. A report, like a letter, is a message with a defined set of message blocks.

The sender of a message can request information about successful delivery of the message, failure to deliver the message, or both, for a message. The sender's request applies to all of the message's recipients.

A single report may contain information about the outcome of delivery attempts to one or more recipients of a message; that is, it may contain delivery indications, non-delivery indications, or both. A ***delivery indication*** indicates the successful delivery of a specific message to one or more specified recipients. A ***non-delivery indication*** indicates failure to deliver a specific message to one or more specified recipients. A delivery or non-delivery indication is sometimes referred to as a *recipient report*.

An MSAM can both create a report about an outgoing message and receive a report about an incoming message.

Note
A report that an MSAM creates or receives (an MSAM report) differs somewhat from a report created or received by other clients of the IPM Manager (an IPM report). An IPM report may contain a copy of the original message, but an MSAM report never does. An IPM report goes directly to an IPM Manager client. An MSAM report goes to an AOCE agent, which interprets the information in the MSAM report and creates an IPM report to send to the ultimate report recipient. ◆

The sections "Generating a Report" on page 2-61 and "Receiving a Report" on page 2-80 describe how an MSAM generates and receives reports. For information on IPM reports, see the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces*.

AOCE Addresses

The AOCE software architecture provides for the exchange of messages among different types of messaging systems. The exchange of messages requires a way of uniquely specifying the sender and receiver of a message. This unique specification is called an *address*. This section discusses the syntax and semantics of the AOCE address structure.

To provide connectivity between AOCE messaging systems and other messaging systems, the AOCE address structure is designed to accommodate already existing address formats, in addition to address formats that may be developed for future messaging systems.

One way that messaging systems can be differentiated is by the syntax and semantics of their addresses. Messaging systems that share the same addressing conventions are said to be of the same type.

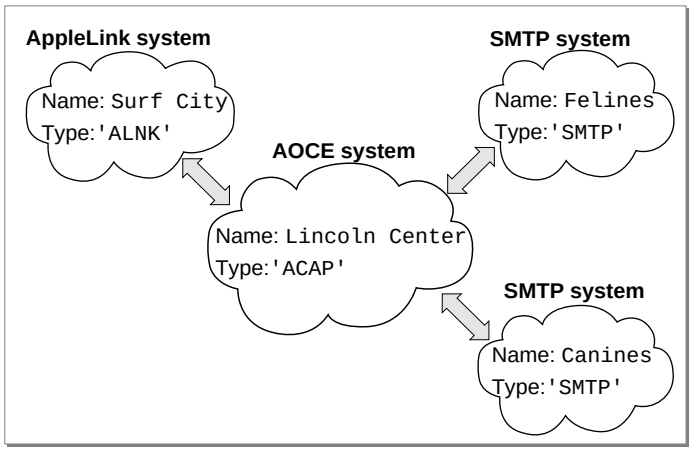
An address is unique within a messaging system. To exchange messages between messaging systems, a sender must specify an address plus the messaging system in which the address is unique.

At the most general level, you can think of an AOCE address structure as having two parts: a messaging system specifier and an entity specifier that uniquely identifies a person or process within that messaging system. When an address specifies a recipient within an AOCE messaging system, the AOCE software delivers the message to the specific address. When an address specifies a recipient in a non-AOCE messaging system, the AOCE software delivers the message to the MSAM responsible for that messaging system.

For AOCE routing software, the basic problem can be stated as follows: assume an external messaging system is named *System X*. System X contains many addressable entities (users and processes). To send a message to an entity Y in System X, AOCE needs a way to say "Y in System X." AOCE doesn't care what Y is. Y is internal to, and should be unique in, System X.

Figure 2-10 shows an AOCE messaging system, an AppleLink system, and two SMTP systems. (SMTP stands for ***Simple Mail Transfer Protocol.*** Computers connected to the Internet often use SMTP to exchange messages.) Within this environment, AOCE routing software needs a way to specify each messaging system. Each messaging system is partially described by a four-character extension type. An ***extension type*** identifies a type of messaging system that uses a specific addressing convention—for example, an AppleLink system or an X.400 system. Because there can be more than one messaging system of a given type, an address based on the extension type alone is not sufficient to distinguish between two or more messaging systems of the same type. In the illustration, AOCE routing software could not distinguish between the two SMTP systems on the basis of type. To solve this problem, AOCE software requires that each messaging system have a unique name by which it is known within an AOCE system. In Figure 2-10, the names Felines and Canines distinguish between the two SMTP messaging systems.

Figure 2-10 AOCE system connected to external messaging systems



In some cases, there is only one messaging system of a given type, and the messaging system already has a unique, well-known name. The Internet is a good example of this. In cases like this, if your MSAM provides a preassigned name, it should use the well-known name. A unique name for each messaging system is fundamental to AOCE addressing.

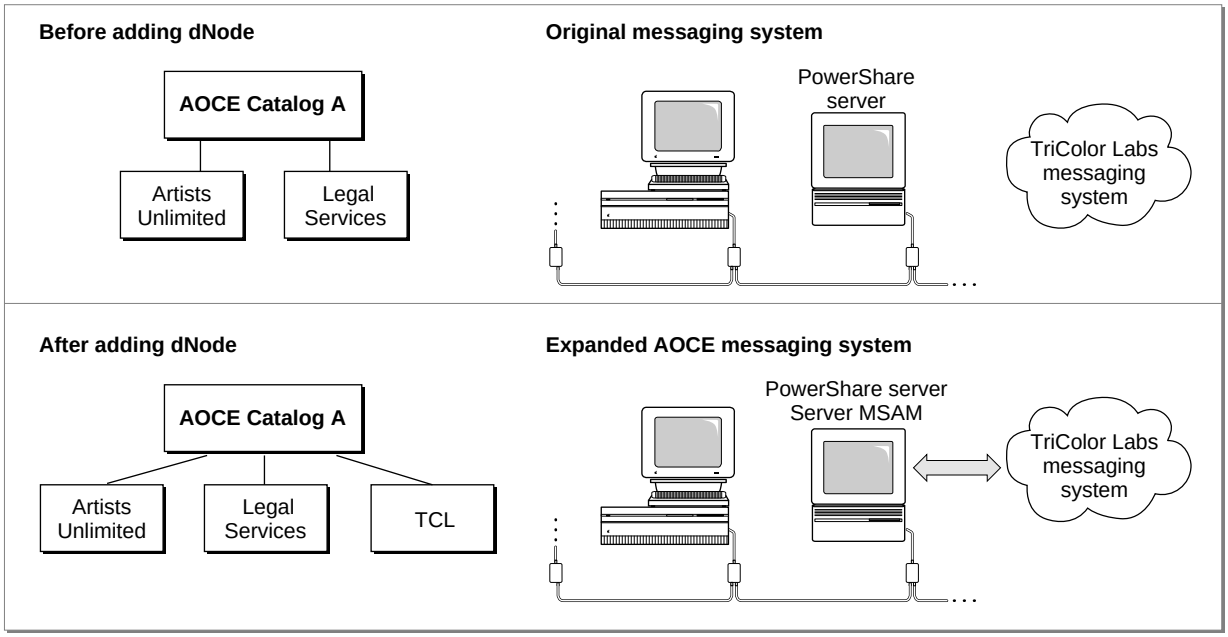
Within some messaging systems, multiple address formats are allowed. The Internet, for example, accepts both UUCP and SMTP addresses. An Internet MSAM has one unique name associated with it, but it may service multiple extension types, one for each form of Internet address that it knows how to translate.

Note
There is no registry for extension types. If you want to use an existing extension type, you are responsible for ensuring that the extension type always represents the same address syntax and semantics. If you want to create a new extension type, it is recommended that you use your application's signature type, registered with Macintosh Developer Technical Services, to ensure uniqueness. ◆

Before describing an AOCE address structure, it is helpful to understand a little about how the AOCE software implements unique names for messaging systems. Within an AOCE system, each external messaging system is associated with a unique catalog name. The catalog name identifies to AOCE software the messaging system and the set of addresses that belong to that messaging system.

For server MSAMs, the AOCE system administrator creates a reference to an external messaging system by creating a dNode, sometimes called a ***foreign dNode,*** in an AOCE catalog. Figure 2-11 illustrates the addition of a dNode that represents an external messaging system. The original AOCE configuration has a catalog named Catalog A that contains dNodes named Artists Unlimited and Legal Services. AOCE software routes messages only among addresses in Catalog A. There exists an external messaging system called TriColor Labs. People within the original AOCE messaging system may want to communicate with people who are accessible only via the TriColor Labs messaging system. A server MSAM is installed within the AOCE system to extend the messaging environment to include people within the TriColor Labs messaging system. The AOCE system administrator creates a new dNode representing the TriColor Labs system and gives the dNode a unique name, TCL, within Catalog A. AOCE software still routes messages only among addresses in Catalog A, but Catalog A now includes a new set of addresses represented by the dNode TCL.

Figure 2-11 Adding a dNode for a messaging system



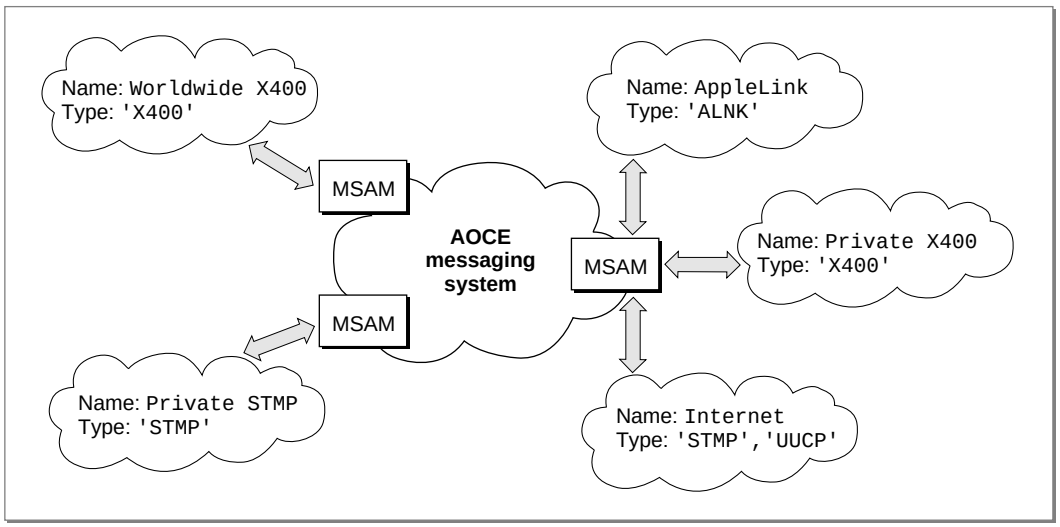
For personal MSAMs, the PowerTalk Key Chain creates a Catalog record in the Setup catalog to represent the set of addresses belonging to a given messaging system. See the chapter “Service Access Module Setup” for more information.

The name that uniquely identifies an external messaging system in an AOCE system is the name of the dNode (for server MSAMs) or the name of the Catalog record in the Setup catalog (for personal MSAMs).

Figure 2-12 illustrates the following points about MSAMs, messaging system names, and extension types:

- An external messaging system must have a unique name.
- Different MSAMs may connect to different external messaging systems of the same extension type.
- A single MSAM may connect to more than one external messaging system, each having a different extension type (it may also connect to more than one external messaging system having the same extension type).
- A single external messaging system may have more than one extension type.

Figure 2-12 MSAMs, messaging system names, and extension types



Now look at the AOCE address structure. AOCE software already defines a `RecordID` structure to uniquely identify a record in an AOCE catalog. This structure is adapted and extended for use as an address structure. In an AOCE messaging system, an address is specified as an `OCERecipient` structure, which is identical to a `DSSpec` structure.

```
struct DSSpec {
   RecordID          *entitySpecifier;
   OSType            extensionType;
   unsigned short    extensionSize;
   Ptr               extensionValue;
   };

typedef DSSpec OCERecipient;
```

(The `RecordID` structure is described in the chapter "AOCE Utilities" in *Inside Macintosh: AOCE Application Interfaces*.)

Figure 2-13 shows an exploded view of an `OCERecipient` structure. An AOCE address is a two-level specification that first identifies a messaging system and then identifies an individual entity within it. This is roughly analogous to an address on a piece of hard-copy mail that specifies a large organization and a mailstop within it. The postal service uses part of the address—organization name, street number, city, state, and zip code—to deliver the mail to the organization. The organization itself uses the remainder of the address, the mailstop number, to deliver the mail to a specific internal address. With an AOCE address, the `OCERecipient.entitySpecifier.rli` substructure identifies the messaging system. The value pointed to by the `OCERecipient.extensionValue` field identifies the individual entity within that messaging system.

Figure 2-13 Exploded view of an `OCERecipient` structure

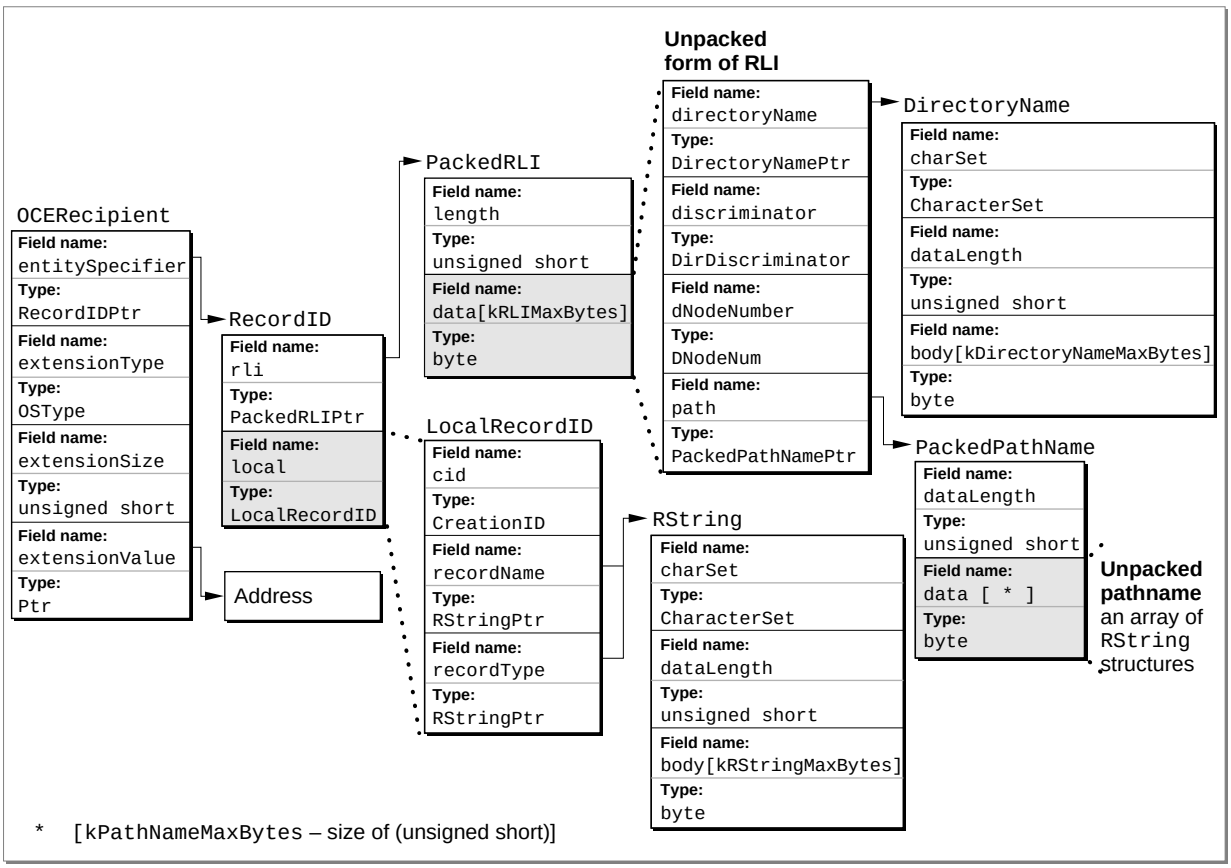


Table 2-4 lists the elemental fields of the address structure and the type of information each field contains when it is used to specify an address on an external system. The structure identifies both the external system and a specific sender or receiver within it that is the source or destination of a message.

Table 2-4 External address: Contents of an `OCERecipient` structure

Field name	Contents
`directoryName`	A pointer to an `RString` structure containing the unique name of a catalog in the AOCE environment. The name identifies the external messaging system to AOCE. The name is limited to 32 characters.
`discriminator`	An 8-byte value that further describes the catalog. The first 4 bytes indicate the extension type of the associated messaging system, for example, ALNK or SMTP. It is the same as the value in the `extensionType` field. The second 4 bytes are private to the catalog.
`dNodeNumber`	Unused. Set to 0.
`path`	Unused. Set to `nil`.
`cid`	Unused. Set to 0.
`recordName`	A pointer to an `RString` structure containing the name of the sender or receiver. This should be a displayable string.
`recordType`	A pointer to an `RString` structure containing the type of the sender or receiver—for example, “user” or “group”. This should be a displayable string.
`extensionType`	The four-character extension type that specifies a type of messaging system, for example, `'ALNK'` or `'SMTP'`. The extension type is the same as the first 4 bytes of the associated catalog’s discriminator value.
`extensionSize`	The length, in bytes, of the `extensionValue` field.
`extensionValue`	A pointer to a buffer that contains the address of the sender or receiver on the external system. The address is used only by the MSAM. Its content and format are not examined by AOCE software. However, for the type-in addressing feature in the mailer to work, the address must be a single `RString` structure.

Table 2-5 lists the elemental fields of the `OCERecipient` structure and the type of information each field contains when it is used to specify an address within an AOCE system.

Table 2-5 AOCE address: Contents of an `OCERecipient` structure

Field name	Contents
`directoryName`	A pointer to an `RString` structure containing the name of the PowerShare catalog that contains the record representing the sender or receiver. The name is limited to 32 characters.
`discriminator`	The discriminator value of the catalog that contains the record representing the sender or receiver.
`dNodeNumber`	A value that identifies the dNode that contains the record representing the sender or receiver. Set to 0 if you use the `path` field to specify the dNode.
`path`	A pointer to a buffer that contains the names of all of the dNodes on the path from the catalog node in which the sender or receiver record resides, up to the catalog root node. Set this field to `nil` if you use the `dNodeNumber` field to identify the dNode.
`cid`	The creation ID of the record that represents the sender or receiver.
`recordName`	A pointer to an `RString` structure containing the name of the sender or receiver. This is a displayable string.
`recordType`	A pointer to an `RString` structure containing the type of the sender or receiver. It tells you what the entity is, such as a user. This is a displayable string.
`extensionType`	A four-character extension type that specifies the format of the data pointed to by the `extensionValue` field. AOCE defines the following extension types: `kOCEalanXtn`, `kOCEentnXtn`, `kOCEaphnXtn`.
`extensionSize`	The length, in bytes, of the `extensionValue` field.
`extensionValue`	A pointer to a buffer that contains the address of the sender or receiver on the AOCE system. The address is used only by the AOCE software. Its content and format need not be examined by the MSAM.

Table 2-6 lists the extension types for addresses within an AOCE messaging system. These extension types are discussed in more detail in the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces*. You do not need to understand the semantics of the extension types. You do need to be sure that a recipient to whom you transmit a message from an AOCE system can reply to the message. Your MSAM might include the extension information with the outgoing message and reconstruct it when it submits the reply to the AOCE system. Alternatively,

your MSAM might maintain mapping tables to convert between addresses within the AOCE messaging system and external addresses. In this way, it can avoid sending to its external system information that is only relevant inside an AOCE system. This implementation decision is up to you.

Table 2-6 AOCE extension types

Constant	Value	Description
`kOCEalanXtn`	`'alan'`	Indicates an `EntityName` structure (an NBP name) plus a queue name in the form of a Pascal string. It is used for an address accessible on the local AppleTalk network.
`kOCEentnXtn`	`'entn'`	Indicates a `DSSpec` structure. It is used for an address accessible through a PowerShare mail server.
`kOCEaphnXtn`	`'aphn'`	Indicates a structure that specifies an address accessible by telephone.

Before you submit an incoming message to AOCE, you must construct `OCERecipient` structures containing the addresses of the sender and each of the recipients. Table 2-4 on page 2-29 describes the information you must provide in each field of the address structure for (a) the sender from your external messaging system and (b) any recipient in an external messaging system. Table 2-5 on page 2-30 describes the information you must provide in each field of the address structure for a recipient within the AOCE system.

When you read an outgoing message from AOCE, you must translate the `OCERecipient` structures that contain the address information for the sender and each of the recipients into a format that your external messaging system understands. Table 2-4 on page 2-29 describes what information you will find in each field of the address structure when the structure specifies a recipient on an external messaging system. Table 2-5 on page 2-30 describes the information contained in each field of the address structure when the structure specifies the sender of an outgoing message or a PowerTalk recipient.

The address of a recipient in an AOCE messaging system might include only the entity specifier portion of the `OCERecipient` structure; that is, it may not have any data in the `extensionType`, `extensionSize`, and `extensionValue` fields. This form is called an *indirect address* because it is not actually an address but points to a record in an AOCE catalog that contains the address. It uniquely identifies the messaging system and provides a displayable name and type to identify the sender or receiver. The direct form of an address always includes both the entity specifier and the extension information. The extension information gives a more detailed form of address. Addresses in external messaging systems are always in the direct form. Addresses in PowerShare catalogs may be in either the direct or indirect form. For more information about direct and indirect addressing, see the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces*.

Table 2-7 shows examples of the content of the fields of an `OCERecipient` structure for an indirect AOCE address and an SMTP address.

Table 2-7 Sample addresses

`OCERecipient` fields	AOCE system (indirect address form)	SMTP system
`directoryName`	Engineering	Finance
`discriminator`	ACAP1234	SMTP0000
`dNodeNumber`	6	0
`path`	`nil`	`nil`
`creationID`	44894489	00000000
`recordName`	Joe Bernard	Suzy Durksen
`recordType`	`aoce User`	`aoce User`
`extensionType`	Not applicable	'`SMTP`'
`extensionSize`	Not applicable	16
`extensionValue`	Not applicable	Suzy@finance.com

When the `entitySpecifier` portion of the `OCERecipient` structure contains information about a sender or receiver on an external system, that information does not specify a record in a PowerShare catalog that represents the sender or receiver. However, when the structure contains information on a sender or receiver inside an AOCE messaging system, it does specify an existing record.

With your MSAM, you need to provide a special kind of AOCE template, called an *address template*, that allows a user to enter address information. Basic information about AOCE templates is provided in the chapter "AOCE Templates" in *Inside Macintosh: AOCE Application Interfaces*. Specific information about address templates is provided in the chapter "Service Access Module Setup" in this book.

AOCE High-Level Events

Both personal and server MSAMs must be prepared to receive and respond to high-level events defined by AOCE software. The chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* describes the use of high-level events in detail; that information is not repeated in this section.

Personal MSAMs may receive the following high-level events:

Constant	Event ID	Description
kMailEPPCCreateSlot	'crsl'	Slot created
kMailEPPCModifySlot	'mdsl'	Slot modified
kMailEPPCDeleteSlot	'dlsl'	Slot deleted
kMailEPPCMailboxOpened	'mbop'	User opened mailbox
kMailEPPCMailboxClosed	'mbcl'	User closed mailbox
kMailEPPCMsgPending	'msgp'	Messages waiting to be sent
kMailEPPCSendImmediate	'sndi'	Send letter now
kMailEPPCShutDown	'quit'	Shut down operations and quit
kMailEPPCContinue	'cont'	Resume operation after error fixed
kMailEPPCSchedule	'sked'	Time for scheduled activity
kMailEPPCInQUpdate	'inqu'	Incoming queue updated
kMailEPPCMsgOpened	'msgo'	User opened letter
kMailEPPCDeleteOutQMsg	'dlom'	Delete outgoing queue message
kMailEPPCWakeup	'wkup'	Launched due to wakeup
kMailEPPCLocationChanged	'locc'	System location changed

Server MSAMs may receive these high-level events:

Constant	Event ID	Description
kMailEPPCAdmin	'admn'	Server administration function
kMailEPPCMsgPending	'msgp'	Messages waiting to be sent

Detailed descriptions of these events can be found in the section "High-Level Events" beginning on page 2-220.

When an MSAM receives an AOCE high-level event, it manipulates a standard `EventRecord` structure (defined in the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*). The fields of an event record associated with an AOCE high-level event have a particular meaning.

```
struct EventRecord {
   short       what;
   long        message;
   long        when;
   long        where;
   short       modifiers;
};
```

Field descriptions

`what` Always contains the constant `kHighLevelEvent`.

`message` Always contains the event class `kMailAppleMailCreator`.

`when`	Unused.
`where`	Contains the event ID that identifies a specific event—for example, `kMailEPPCAdmin`.
`modifiers`	For personal MSAMs, this field contains the slot ID when the event applies to a particular slot; otherwise, it is set to 0. Server MSAMs can ignore this field.

Some AOCE high-level events require more information than that provided in the event record. After you receive such an event, you should call the `AcceptHighLevelEvent` function to get the additional data associated with the event. The additional data is in the form of a `MailEPPCMsg` structure.

A `MailEPPCMsg` structure consists of a version number and a union field. The union field may have any of the following contents: a pointer to an `SMCA` structure; a letter sequence number; a `MailLocationInfo` structure.

The version number indicates the version of the event. The MSAM should compare the version number in the `MailEPPCMsg` structure with `kMailEPPCMsgVersion`. If they are not the same, software incompatibilities may exist between the PowerTalk software and the MSAM, and there is no guarantee that the `MailEPPCMsg` structure used by the MSAM and by the IPM Manager are the same. The MSAM should ignore the event.

Most of the AOCE high-level events are informational in nature. For example, a `kMailEPPCMsgPending` event tells an MSAM that it has a new outgoing message. Informational events sent by the IPM Manager are not guaranteed to be received by the MSAM. The MSAM should consider these events as hints; that is, it should not rely on them as the only mechanism to initiate an action. For example, to make sure it transfers outgoing messages in a timely manner, it could check its outgoing queues every 20 minutes, each time it is launched, and each time it receives a `kMailEPPCMsgPending` event.

A few events are more than informational in nature. An MSAM must receive the `kMailEPPCCreateSlot`, `kMailEPPCModifySlot`, `kMailEPPCDeleteSlot`, `kMailEPPCMsgOpened`, and `kMailEPPCSendImmediate` events in order to take the relevant actions. For these events, the `MailEPPCMsg` structure contains a pointer to an `SMCA` structure. The MSAM needs to set the `result` field of the `SMCA` structure to acknowledge the event or to report the outcome of its effort to handle the event. Additionally, the IPM Manager informs the client if the event does not reach the MSAM. (An MSAM cannot acknowledge or set a result for an event whose `MailEPPCMsg` structure does not contain a pointer to an `SMCA` structure.)

Once the MSAM sets the `result` field to acknowledge the event or to signal completion, the `SMCA` structure is no longer valid.

An MSAM defines the error codes that it returns in response to the `kMailEPPCCreateSlot`, `kMailEPPCModifySlot`, `kMailEPPCDeleteSlot`, and `kMailEPPCMsgOpened` events. For the `kMailEPPCSendImmediate` event, it typically should return the `kMailSlotSuspended` or `kMailTooManyErr` result code.

System Location

The concept of location serves users with mobile Macintosh computers. Personal MSAMs must understand the concept of location, whereas server MSAMs need not. A personal MSAM, residing on a user's Macintosh, must be aware of the possibility that the system location may change. For instance, a personal MSAM installed on a PowerBook may be launched at different locations, such as the user's business office, the user's home, a customer site, an airport, and so forth. The personal MSAM is likely to be affected by such changes of location. A fax MSAM, for example, would use different telephone numbers when running at home or in the office; an Internet MSAM cannot work if a TCP/IP network connection is not available.

After it is launched, a personal MSAM gets the current system location from the Setup record in the Setup catalog. Then it determines, for each slot, whether the slot is active at that location by checking the location flags in the slot's standard slot information. See the section "Initializing a Personal MSAM" on page 2-37 for a description of how you do this.

If a slot is not active at the current location, the personal MSAM should not perform any work on behalf of that slot. If none of the personal MSAM's slots are active at the current location, the MSAM should quit.

If the system location changes, the IPM Manager sends the MSAM one `kMailEPPCLocationChanged` high-level event for each slot. The event tells the MSAM the slot to which it applies, the current system location, and the location flags for the slot. If the location flags show that the slot is inactive at the current location, the MSAM should immediately stop performing any activity on behalf of the slot, such as downloading or sending letters.

A user can activate or deactivate a mail slot in a given location. In response, the IPM Manager updates the location flags in the `MailStandardSlotInfoAttribute` structure for that slot and sends a `kMailEPPCLocationChanged` high-level event to the MSAM. At that point, the MSAM needs to determine if the slot is active at the current location. If the slot is active, the MSAM should continue to act for the slot; if it is not, the MSAM should cease acting for the slot.

Using the MSAM API

This section shows you how to

- determine whether the Collaboration toolbox is available
- launch a personal MSAM
- initialize personal and server MSAMs
- transfer an outgoing letter from an AOCE system to another messaging system

- transfer an incoming letter from another messaging system to an AOCE system
- delete a message
- translate addresses
- log personal MSAM operation errors

Determining Whether the Collaboration Toolbox Is Available

Before calling any of the functions in the MSAM API, a server MSAM should verify that the Collaboration toolbox is available by calling the `Gestalt` function with the selector `gestaltOCEToolboxAttr`. If the Collaboration toolbox is present but not running (for example, if the user deactivated it from the PowerTalk Setup control panel), the `Gestalt` function sets the bit `gestaltOCETBPresent` in the `response` parameter. If the Collaboration toolbox is running and available, the function sets the bit `gestaltOCETBAvailable` in the `response` parameter. The Gestalt Manager is described in the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*. Because a personal MSAM is launched by the IPM Manager, it can assume that the Collaboration toolbox is available.

If you want to be informed when the IPM Manager starts up or shuts down, you can install an entry in the AppleTalk Transition Queue (ATQ). Then the AppleTalk Link-Access Protocol Manager calls your ATQ routine with the transition selector `ATTransIPMStart` when the IPM Manager has finished starting up and with the selector `ATTransIPMShutdown` when the IPM Manager has started to shut down. The ATQ is described in the "Link-Access Protocol (LAP) Manager" chapter in *Inside Macintosh: Networking*.

Determining the Version of the IPM Manager

To determine the version of the IPM Manager that is available, call the `Gestalt` function with the selector `gestaltOCEToolboxVersion`. The function returns the version number of the Collaboration toolbox in the low-order word of the `response` parameter. For example, a value of 0x0101 indicates version 1.0.1. If the Collaboration toolbox is not present and available, the `Gestalt` function returns 0 for the version number. You can use the constant `gestaltOCETB` for AOCE Collaboration toolbox version 1.0.

Launching a Personal MSAM

A personal MSAM must be launched by the IPM Manager. If you launch a personal MSAM in any other manner, it will not work properly with the IPM Manager.

If a personal MSAM is not already running, the IPM Manager launches it in response to any of the following events:

- The MSAM's setup template calls the `MailCreateMailSlot` or `MailModifyMailSlot` function.
- An application calls the `MailWakeupPMSAM` function.

- The MSAM's scheduled send or receive time occurs, or its send/receive time interval elapses.

Initializing a Personal MSAM

Before the IPM Manager launches a personal MSAM for the first time, the setup template you provide with your personal MSAM must obtain information about the MSAM, the accounts on external messaging systems to which it will connect, and the catalogs associated with those external messaging systems. It gets this information from the user and stores it in the Setup catalog.

Once launched, a personal MSAM needs to obtain a variety of information, much of it in the Setup catalog. The information includes:

- the current system location
- information about each slot for which it is responsible (each slot represents one account on a messaging system)
- the incoming and outgoing queue references for each of its slots
- any additional configuration or private information it may require

A personal MSAM obtains much of the necessary information by reading records in the Setup catalog. It then often copies this information into private structures.

The following steps illustrate a typical sequence of actions your MSAM can take to obtain the necessary startup information after it has been launched:

1. Get the creation ID of the MSAM's record in the Setup catalog by calling the `PMSAMGetMSAMRecord` function. Build a record ID that contains your MSAM's record creation ID.
2. Get the local identity by calling the `AuthGetLocalIdentity` function. If the user hasn't set up a local identity yet, the function returns the `kOCESetupRequired` result code. If the local identity is locked, the function returns the `kOCELocalAuthenticationFail` result code. In either case, call the `AuthAddToLocalIdentityQueue` function to be notified when the local identity is set up and unlocked. If the `AuthGetLocalIdentity` function returned `kOCELocalAuthenticationFail`, you can pass the locked local identity provided by the function to the `DirLookupGet` and `DirLookupParse` functions. Therefore, you should proceed with the initialization process.
3. Get the reference number of the Setup catalog and the creation ID of the Setup record by calling the `DirGetOCESetupRefnum` function. You need to provide the catalog's reference number in the `dsRefNum` field of the `DirLookupGet` and `DirLookupParse` parameter blocks when you want to read the records in the Setup catalog. You need the creation ID to build a record ID for the Setup record.
4. Get the current location from the Setup record in the Setup catalog by calling the `DirLookupGet` and `DirLookupParse` functions. As the target of the `aRecordList` field in the `DirLookupGet` parameter block, specify the record ID of the Setup record. You can set all fields of the record ID except the creation ID to `nil`. Set the creation ID to the value you obtained in the previous step. Instead of providing record location information, you provide the catalog's reference number in the `dsRefNum` field of the

DirLookupGet function's parameter block. As the target of the attrTypeList field in the parameter block, specify the AttributeType structure referenced by the attribute type index kLocationAttrTypeNum. The function reads the Setup record and places the location information into a buffer in a private data format.

Call the DirLookupParse function to read the data in the buffer. The function calls a callback routine that you provide and passes it a pointer to an Attribute structure containing the location information (type OCESetupLocation) that you requested.

5. Get a reference to each Mail Service or Combined record that belongs to the MSAM by calling the DirLookupGet and DirLookupParse functions. If you provide a stand-alone MSAM, attributes for a slot and its associated catalog are stored in a Mail Service and a Catalog record, respectively. If you provide a combined MSAM/CSAM, attributes for a slot and its associated catalog are stored in a single Combined record.

   As the target of the aRecordList field in the DirLookupGet parameter block, specify the RecordID structure that you created that contains the creation ID of your MSAM record. As the target of the attrTypeList field in the parameter block, specify the AttributeType structure referenced by the attribute type index kMailServiceAttrTypeNum. The function reads the MSAM record and places the packed record ID of each Mail Service or Combined record that it finds into a buffer in a private data format.

   Call the DirLookupParse function to read the data in the buffer. The function calls your callback routine and passes it a pointer to an Attribute structure containing a packed record ID that points to either a Mail Service or a Combined record. The DirLookupParse function calls your callback routine once for each packed record ID in the buffer, each of which corresponds to a slot for which your MSAM is responsible. Now you know how many slots you are responsible for and in what records their specific information is stored.

6. Unpack the packed record IDs of the Mail Service or Combined records by calling the OCEUnpackRecordID utility function.

7. Get the slot ID, standard slot information, and associated catalog information for each slot by calling the DirLookupGet and DirLookupParse functions. As the target of the aRecordList field in the DirLookupGet parameter block, specify the unpacked record IDs that point to your Mail Service or Combined records. As the target of the attrTypeList field in the parameter block, specify AttributeType structures that are referenced by the following attribute type indexes: kSlotIDAttrTypeNum, kStdSlotInfoAttrTypeNum, and kAssoDirectoryAttrTypeNum.

   Call the DirLookupParse function. It repeatedly calls your callback routine and passes it a pointer to an Attribute structure containing one of the record attributes you requested for each of your Mail Service or Combined records.

   The value of each kSlotIDAttrTypeNum attribute is the slot ID you previously assigned to the slot while processing the kMailEPPCCreateSlot high-level event for that slot. It is a number (type MailSlotID) that uniquely identifies the slot. (If you have never received and processed a kMailEPPCCreateSlot high-level event, no kSlotIDAttrTypeNum attributes exist.)

The value of each `kStdSlotInfoAttrTypeNum` attribute is a `MailStandardSlotInfoAttribute` structure that indicates if the slot is active and provides its send and receive timer information. For each slot, you must determine if the slot is active at the current system location. The `active` field of the `MailStandardSlotInfoAttribute` structure is a bit array; each bit corresponds to a possible system location. If the slot is active at that location, the bit is set. You can test the bits with the `MailLocationMask` macro (see page 2-115)

The value of each `kAssoDirectoryAttrTypeNum` attribute is a packed record ID that points to the Catalog record associated with this slot or to the Combined record.

8. If you provide a stand-alone MSAM, unpack the packed record ID for each slot's associated Catalog record by calling the `OCEUnpackRecordID` utility function. (If you provide a combined MSAM/CSAM, attributes for the slot and catalog are both stored in the Combined record— you already unpacked the Combined record IDs.)
9. Get information about the catalog associated with each slot by calling the `DirLookupGet` and `DirLookupParse` functions. As the target of the `aRecordList` field in the `DirLookupGet` parameter block, specify the unpacked record IDs that point to your Catalog or Combined records. As the target of the `attrTypeList` field in the parameter block, specify `AttributeType` structures that are referenced by the following attribute type indexes: `kCommentAttrTypeNum`, `kRealNameAttrTypeNum`, and `kDiscriminatorAttrTypeNum`. If you provide a combined MSAM/CSAM, also specify `kSFlagsAttrTypeNum`.

   Call the `DirLookupParse` function. It repeatedly calls your callback routine and passes it a pointer to an `Attribute` structure containing one of the attributes you requested from each Catalog or Combined record. Table 2-8 on page 2-40 describes the information contained in those attributes.
10. Get the user's account name and decrypted password by calling the `OCESetupGetDirectoryInfo` function. If the local identity is still locked, this function returns an error. You cannot proceed until the local identity is unlocked.

    Note that the value of the `nativeName` field returned by the `OCESetupGetDirectoryInfo` function is the value of the Real Name attribute (`kRealNameAttrTypeNum`) in the Catalog or Combined record. The content and use of the Real Name attribute and the `nativeName` field are defined by the personal MSAM and its setup template. A setup template can store the user's account name in the Real Name attribute.

    At this point, you have obtained all of the standard information stored in your MSAM and Combined records (or MSAM, Mail Service, and Catalog records) in the Setup catalog. Using the `DirLookupGet` and `DirLookupParse` functions, you may read other attributes of private types that your setup or address template has added to the records.
11. Get the incoming and outgoing queue references for each of the slots by calling the `PMSAMOpenQueues` function for each slot.

Now the personal MSAM can begin performing its primary functions of translating and transferring messages between an AOCE system and external messaging systems.

Table 2-8 Selected Catalog record attributes

Attribute type	Data type of attribute value	Description
`kDiscriminatorAttrTypeNum`	`DirDiscriminator`	Discriminator value for this catalog.
`kSFlagsAttrTypeNum`	`long`	Bit array indicating the features supported by this catalog. Present for combined MSAM/CSAM only.
`kCommentAttrTypeNum`	`RString`	Displayable string describing this catalog/external messaging system.
`kRealNameAttrTypeNum`	`RString`	Defined by the MSAM and its setup template. For example, it may be the user's account (logon) name or the name of the external messaging system and its address catalog.

The chapter "Service Access Module Setup" in this book describes the information that your setup template obtains from the user and stores in the Setup catalog as well as the process it uses to do so. See the chapter "Catalog Manager" in *Inside Macintosh: AOCE Application Interfaces* for descriptions of the `DirGetOCESetupRefnum`, `DirLookupGet`, and `DirLookupParse` functions. For a description of the `OCEUnpackRecordID` function and the record and attribute type indexes, see the chapter "AOCE Utilities" in *Inside Macintosh: AOCE Application Interfaces*. The `OCESetupGetDirectoryInfo` function is described in the chapter "Authentication Manager" in *Inside Macintosh: AOCE Application Interfaces*.

Initializing a Server MSAM

The first time a server MSAM is launched, it needs to solicit user input to obtain information about itself. Then it initializes itself within the AOCE system by calling the `SMSAMSetup` and `SMSAMStartup` functions.

The `SMSAMSetup` function creates the server MSAM's Forwarder record. The ***Forwarder record*** (record type index `kMnMForwarderRecTypeNum`) contains information about the server MSAM. The Forwarder record name is the name of the server MSAM. The record contains the record ID of the MSAM's PowerShare mail server, an optional comment string describing the server MSAM, and a list of the foreign dNodes to which the server MSAM is connected. (See the chapter "Catalog Manager" in *Inside Macintosh: AOCE Application Interfaces* for information about PowerShare catalogs, dNodes, and foreign dNodes, as well as other concepts that pertain to AOCE catalogs.)

After being launched for the first time, a server MSAM must find out its name, password, messaging system extension type, and a descriptive comment string about the extension type. The MSAM should display one or more dialog boxes to obtain its name and password from the system administrator. Typically, an MSAM has built-in knowledge of the extension type it supports and a descriptive comment string about the extension type; if it does not, it must obtain that information from the system administrator.

Once a server MSAM has all this information, it calls the `SMSAMSetup` function to create its Forwarder record. Prior to calling the function, the MSAM must allocate a `RecordID` structure for its Forwarder record. Then the MSAM sets the `recordName` field to its name that the user provided, and the `recordType` field to the constant `kMnMForwarderRecTypeNum`. The MSAM passes to the function a pointer to the `RecordID` structure, the MSAM's password, its extension type, and a string describing its extension type. In the `RecordID` structure, the function returns the creation ID for the newly created Forwarder record and the record location information. In the `catalogServerHint` field, the function returns the AppleTalk address (an `AddrBlock` structure) of the PowerShare catalog server that created the Forwarder record. The MSAM can pass this address to a Catalog Manager function (in the `serverHint` field of the function's parameter block) if it wants to direct the request to that particular catalog server. This can be helpful in preventing failures in the setup process due to delays in replicating the MSAM's Forwarder record.

During the execution of the `SMSAMSetup` function, the PowerShare mail server prompts the user for the system administrator's name and password. You may find it helpful to consult the *PowerShare System Manager's Guide*, which describes the setup process from the system administrator's perspective.

If the system administrator does not provide this information, the function returns an error. The function will also return an error if

- the PowerShare catalog server was unreachable
- the MSAM's name is not unique
- the disk is full
- an error occurred in creating the Forwarder record (any record creation error)

If an error occurs, the MSAM must display an appropriate dialog box telling the user about the error. If the PowerShare catalog server was unreachable, the MSAM should give the user the option of trying the operation again and, if the user chooses to try again, the MSAM should call the `SMSAMSetup` function once more. If the user chooses not to try again, the MSAM should quit. If the MSAM's name was not unique, the MSAM should allow the user to enter another name. In any error, the MSAM should fix the problem when it can or quit when it cannot. Until the `SMSAMSetup` function executes successfully, the MSAM cannot proceed with its initialization process.

When the `SMSAMSetup` function completes successfully, the server MSAM must save knowledge of this fact so that if it is launched again in the future, it does not call the `SMSAMSetup` function again. It is recommended that the server MSAM create a preferences file in the Preferences folder and save the record ID of its Forwarder record in its preferences file.

Once the `SMSAMSetup` function completes successfully, the server MSAM should call the `AuthBindSpecificIdentity` function, providing the record ID of its Forwarder record and its encrypted password, to obtain its authentication identity. Once a server MSAM has obtained its authentication identity, it should provide that information on subsequent calls to AOCE functions that require an identity.

At this point, the server MSAM may present dialog boxes to the user to obtain any additional configuration information it needs to function within an AOCE system and to connect to its external messaging system, such as an IP address, a telephone number, how often it should connect, and so forth. In general, an MSAM should ask for more generic information first—that is, information that applies independently of a messaging system. Then it should prompt for specific information for each messaging system that it supports. It should then add this information to its Forwarder record in MSAM-defined attribute types.

Note
In addition to its Forwarder record, a server MSAM should store a copy of its configuration information in its preferences file for quick, efficient access.

A server MSAM should keep a backup copy of its preferences file in case the file is lost or damaged. If its preferences file is lost or damaged and a server MSAM does not have a backup copy, it can retrieve the information stored in the MSAM's Forwarder record and rebuild the file. To read its Forwarder record, an MSAM must have the Forwarder record ID (which it obtains from the `SMSAMSetup` function). ◆

As the final step in the server MSAM's initialization process, the MSAM calls the `SMSAMStartup` function to obtain a reference number for its outgoing queue. After the `SMSAMStartup` function completes successfully, the PowerShare mail server may send high-level events to the server MSAM. The MSAM should respond to high-level events, connect to external messaging systems, and begin to translate and transfer messages.

A server MSAM must run on the same Macintosh computer as its PowerShare mail server. If the PowerShare mail server is not running, the `SMSAMStartup` function returns the `corErr` result code. You can detect when the PowerShare mail server becomes available by

- repeatedly calling the `Gestalt` function and using the `gestaltOCESFServerAvailable` mask on its `response` parameter to determine if a PowerShare mail server is running on the local Macintosh computer
- repeatedly calling the `SMSAMStartup` function
- adding an entry to the AppleTalk Transition Queue and waiting to receive a notification that the PowerShare mail server is available

Using the AppleTalk Transition Queue is the recommended approach. The transition event code `ATTransSFStart` indicates that the PowerShare mail server has finished starting up, and the code `ATTransSFShutdown` indicates that the PowerShare mail server has started to shut down.

The AppleTalk Transition Queue is described in the chapter "Link-Access Protocol (LAP) Manager" in *Inside Macintosh: Networking*.

If the PowerShare mail server quits, your queue reference becomes invalid. You know that the PowerShare mail server is not running when any of the MSAM API functions return the `corErr` result code or you receive notification of the `ATTransSFShutdown` AppleTalk transition event. If the PowerShare mail server quits unexpectedly, you do not receive an AppleTalk transition event.

When it starts up again, the PowerShare mail server does not know that your server MSAM exists. You need to call the `SMSAMStartup` function again to get a new queue reference. You detect the restarting of the PowerShare mail server by any of the three methods listed previously.

If the PowerShare mail server quits, your server MSAM can keep running. Although you can no longer retrieve messages from your outgoing queue, you can continue to process any outgoing messages you queued separately. You can mark recipients and send reports for those messages after the PowerShare mail server resumes operations. If you have a separate spool area to hold them, you can continue to process incoming messages while the PowerShare mail server is not running.

Handling Outgoing Messages

This section describes what you need to do with messages in an outgoing queue. It assumes you have already initialized your MSAM. Each subsection addresses a specific task, such as

- enumerating messages in an outgoing queue
- opening and closing messages
- determining the message family
- determining what is in a message
- reading letter attributes
- reading addresses
- reading letter content
- reading nested messages
- marking recipients
- generating reports

There are some differences between how you read letters and how you read non-letter messages. These differences are noted in the sections that address the specific tasks. For convenience, Table 2-9 lists the tasks you perform while handling messages in an outgoing queue and the functions you use to accomplish the task for a letter and a non-letter message.

Table 2-9 Outgoing tasks and functions

Task	Letters	Non-letter messages
Enumerate a queue	`MSAMEnumerate`	`MSAMEnumerate`
Open a message	`MSAMOpen` `MSAMOpenNested`	`MSAMOpen` `MSAMOpenNested`
Read header information	`MSAMGetAttributes` `MSAMGetRecipients`	`MSAMGetMsgHeader` `MSAMGetRecipients`
Read letter content	`MSAMGetContent`	Not applicable
Read an enclosure	`MSAMGetEnclosure`	Not applicable
Enumerate a block	`MSAMEnumerateBlocks`	`MSAMEnumerateBlocks`
Read a block	`MSAMGetBlock`	`MSAMGetBlock`
Close a message	`MSAMClose`	`MSAMClose`

Generate a report	`MSAMCreateReport`
	`MSAMPutRecipientReport`
	`MSAMSubmit`
		`MSAMCreateReport`
		`MSAMPutRecipientReport`
		`MSAMSubmit`

Mark a recipient	`MSAMnMarkRecipients`	`MSAMnMarkRecipients`
Set message status (personal MSAMs only)	`PMSAMSetStatus`	`PMSAMSetStatus`

The order in which functions are listed in Table 2-9 corresponds to the sequence in which you would call the functions to process a message in an outgoing queue. You first enumerate the messages in the queue. Then you open a specific message and read its header information. Header information consists of such items as the message creator and type, and address (recipient) information. Next, you read the substance of the message—for a letter, its content block, other blocks it may contain, and enclosures; for a non-letter message, its blocks. When you have finished reading the message, you close it. After you have transmitted the message to the recipients for which you are responsible, you indicate the outcome of your delivery attempts—that is, you generate a report containing delivery and non-delivery indications if required and mark the recipients. Setting the status of a message is a task that you perform at several points while you are processing the message.

You should call the functions that handle outgoing messages asynchronously so that you can receive and process an AOCE high-level event at any time.

Enumerating Messages in an Outgoing Queue

Before you can read a message from an outgoing queue, you must obtain its sequence number. A sequence number uniquely identifies the message in the queue. You provide it when you open the message. You get the sequence number of a message by calling the `MSAMEnumerate` function.

To make sure it transfers outgoing messages in a timely manner, an MSAM should enumerate an outgoing queue on a regular basis. The MSAM should enumerate each

time it is launched and each time it receives a `kMailEPPCMsgPending` event. It should also enumerate at periodic intervals—for instance, every 20 minutes. If an MSAM puts itself into an idle state, it should enumerate before entering the idle state. A personal MSAM should also enumerate when it receives a `kMailEPPCSchedule` event.

Listing 2-1 illustrates one way that you can enumerate messages in an outgoing queue. For convenience, the function `DoEnumerateOutgoingMessages` in Listing 2-1 defines the type `MyEnumOutQReplyType`, a structure that contains a buffer that can hold a 2-byte count value plus exactly one `MSAMEnumerateOutQReply` structure. As a result, each time `DoEnumerateOutgoingMessages` calls the `MSAMEnumerate` function, `MSAMEnumerate` returns exactly one `MSAMEnumerateOutQReply` structure, which provides identifying information about one message in the queue, including its sequence number.

Before the `DoEnumerateOutgoingMessages` function calls the `MSAMEnumerate` function, it always initializes the fields of the parameter block. It sets the queue reference to an outgoing queue reference previously obtained from the `PMSAMOpenQueues` function. The first time through the loop, `DoEnumerateOutgoingMessages` sets the starting sequence number to 1 to start with the first message in the queue. On subsequent executions of the loop, it sets the starting sequence number to the sequence number of the next message in the queue, which is returned by `MSAMEnumerate`.

The `DoEnumerateOutgoingMessages` function calls the `MSAMEnumerate` function once for each message in the queue. Into your buffer, `MSAMEnumerate` places the count of the number of `MSAMEnumerateOutQReply` structures followed by the reply structures themselves. In Listing 2-1, the count is always 1.

Listing 2-1 Enumerating outgoing messages

```
OSErr DoEnumerateOutgoingMessages(MSAMQueueRef myOutgoingQRef)
{
typedef struct MyEnumOutQReplyType {
   MailReply               reply;   /* number of structures returned */
   MSAMEnumerateOutQReply  message; /* enumerate reply structure */
} MyEnumOutQReplyType;

   OSErr                myErr;
   MSAMEnumeratePB      myParamBlock;
   MyEnumOutQReplyType  myEnumOutQReply;
   long                 myNextMsgSeq;

   myNextMsgSeq   = 1;
   myErr          = noErr;
```

```
    do {
        myParamBlock.ioCompletion        = (ProcPtr)DoMSAMCompletion;
        myParamBlock.queueRef            = myOutgoingQRef;
        myParamBlock.startSeqNum         = myNextMsgSeq;
        myParamBlock.buffer.bufferSize   = sizeof(MyEnumOutQReplyType);
        myParamBlock.buffer.buffer       = (Ptr)&myEnumOutQReply;

        MSAMEnumerate((MSAMParam *)&myParamBlock,true);
        /* poll for completion */
        myErr          = DoWaitPBDone(&myParamBlock);
        myNextMsgSeq   = myParamBlock.nextSeqNum;

        /* save the MSAMEnumerateOutQReply structure */
        DoSaveData((Ptr)&myEnumOutQReply);

    }
    while (myErr == noErr && myNextMsgSeq != 0);

    return myErr;
}
```

The `DoWaitPBDone` function, called here and in the listings in the following sections, polls the `ioResult` field to determine when an asynchronous request has completed. While it is polling, it also yields time to other processes running on the computer by calling the `WaitNextEvent` function. When the `MSAMEnumerate` function completes, `DoWaitPBDone` returns the `MSAMEnumerate` result code as its result code.

The `DoMSAMCompletion` completion routine, called when the `MSAMEnumerate` function completes execution, calls the `WakeUpProcess` function. Then `WakeUpProcess` makes the MSAM process, which suspended itself by calling the `WaitNextEvent` function, eligible to receive CPU time.

After the `MSAMEnumerate` function completes, `DoEnumerateOutgoingMessages` saves the enumeration information elsewhere by calling its `DoSaveData` function. It needs to do this because `MSAMEnumerate` overwrites the `MyEnumOutQReplyType` structure each time through the loop.

Opening and Closing a Message

Before you can read any part of an outgoing message, you must open it. To open a specific message, you call the `MSAMOpen` function and provide the queue reference of the outgoing queue in which the message is located and the sequence number of the message. The `MSAMOpen` function returns a reference number for the opened message that you use when you call other functions to read the various parts of the message, such as the message header, recipient information, and the content data in the message. If the message is a letter, you can also read the letter's attributes. You cannot modify a message in an outgoing queue.

When you have finished reading a message, call the `MSAMClose` function to close it. Closing a message reduces PowerTalk software memory requirements. Once you have closed a message, the message reference number is no longer valid, even though the message itself remains in the outgoing queue. If you want to read any part of the message again, you must call the `MSAMOpen` function and get a new reference number. You can open and close a message as many times as you wish.

Determining the Message Family

You must determine if a message that you want to read is a letter or a non-letter message because the functions you use to read a letter or a non-letter message differ somewhat (see Table 2-9 on page 2-44). You determine the message family to which a message belongs by examining the `msgFamily` field in the `MSAMEnumerateOutQReply` structure. Letters may belong to either the `kMailFamily` or `kMailFamilyFile` family. Non-letter messages belong to the `kIPMFamilyUnspecified` family. Once you know the message family, you can call the appropriate MSAM functions to read the attributes, addresses, and contents of the letter or non-letter message.

Determining What Is in a Message

Typically, when you read a letter, you call the `MSAMGetContent`, `MSAMGetBlock`, `MSAMGetEnclosure`, and `MSAMOpenNested` functions to read the letter's content block, image block, enclosures, and nested letter, respectively.

When you want to read a non-letter message, you need to enumerate the blocks in the message. The `MSAMEnumerateBlocks` function returns each block's creator and type, its offset in bytes from the beginning of the message, and its length in bytes. When you want to read a given block, you call the `MSAMGetBlock` function and provide the block's creator and type.

Reading Letter Attributes

Every letter contains attributes that provide information about the letter, such as whether the sender wants to receive a report containing delivery or non-delivery indications, when the letter was sent, and so forth. You should read this information and include in the letter as much of the information as is meaningful in your messaging system. You can read most letter attributes with the `MSAMGetAttributes` function. However, to read the recipients of a letter—the `from`, `to`, `cc`, and `bcc` attributes—you call the `MSAMGetRecipients` function.

To the `MSAMGetAttributes` function, you provide a set of bit flags, known as the *request mask*, that represents the attributes whose values you want to read and a buffer to hold the attribute values. The `MailAttributeBitmap` structure, described on page 2-100, defines the attributes that the bit flags in the request mask represent. The function returns a second set of bit flags, known as the *response mask*, that indicates which of the requested attribute values it has returned in your buffer.

The function `DoReadLetterAttributes` in Listing 2-2 shows how you can request attribute values, test for their presence in your buffer, and save the value in a file. The `DoReadLetterAttributes` function defines the structure type `MaximumLetterAttributes` that is large enough to hold a value for each of the attributes that the `MSAMGetAttributes` function can return. The `DoReadLetterAttributes` function declares a variable of that type, `myAttribBuf`, and sets a pointer, `myAttribPtr`, to point to the start of the buffer. Next, it initializes the request mask to 0 and then sets the request mask to specify every attribute that the `MSAMGetAttributes` function can return. If the messaging system to which you provide access does not use some of this information, don't ask for it. For instance, if you know that your messaging system does not understand a reply ID, do not set the bit for the reply ID in the attribute request mask.

Note

Because the `MailAttributeBitmap` data type is defined as a bit field structure, you cannot use predefined masks such as `kMailSubjectMask`, `kMailMsgTypeMask`, and so forth to set or test the value of a bit field in a variable of type `MailAttributeBitmap`. The masks operate on variables of type `long`. ◆

After the `DoReadLetterAttributes` function sets its attribute request mask, it calls the `MSAMGetAttributes` function. The `MSAMGetAttributes` function returns the attributes that you request (if they are present in the letter header) packed into your buffer, starting with the attribute specified by the least significant bit in the request mask. The `MSAMGetAttributes` function also sets the bits in the response mask corresponding to those attributes for which it returned a value.

Next, `DoReadLetterAttributes` tests the bits in the response mask to find out which attributes are in the buffer. Initially, `myAttribPtr` points to the beginning of the `myAttribBuf` buffer. For each bit in the response mask that is set, `DoReadLetterAttributes` writes the corresponding attribute value to a file and adds the size of the attribute value's data type to `myAttribPtr` to position the pointer to the start of the next attribute value in `myAttribBuf`.

Listing 2-2 Reading letter attributes

```
OSErr DoReadLetterAttributes(MailMsgRef myMailRef)
{
 /* maximum size structure for calling MSAMGetAttributes */
typedef struct MaximumLetterAttributes {
   MailIndications   indications;
   OCECreatorType    msgType;
   MailLetterID      letterID;
   MailTime          sendTimeStamp;
   MailNestingLevel  nestingLevel;
   OSType            messageFamily;
   MailLetterID      replyID;
```

```
   MailLetterID      conversationID;
   RString           subject;
} MaximumLetterAttributes;

   OSErr                   myErr;
   MSAMGetAttributesPB     myParamBlock;
   MailAttributeBitmap     myRequestBitmap;
   MaximumLetterAttributes myAttribBuf;
   char                    *myAttribPtr;
   long                    *myClearBitmap;


   myAttribPtr = (char *)&myAttribBuf;    /* point to start of buffer */

   /* initialize the request mask to 0 */
   myClearBitmap  = (long *)&myRequestBitmap;
   *myClearBitmap = 0L;

   /* set bits for the attributes you want */
   myRequestBitmap.indications = myRequestBitmap.msgType =
       myRequestBitmap.letterID = myRequestBitmap.sendTimeStamp =
       myRequestBitmap.nestingLevel = myRequestBitmap.msgFamily =
       myRequestBitmap.replyID = myRequestBitmap.conversationID =
       myRequestBitmap.subject = 1;

   /* fill in the fields of the parameter block */
   myParamBlock.ioCompletion     = (ProcPtr)DoMSAMCompletion;
   myParamBlock.mailMsgRef       = myMailRef;
   myParamBlock.requestMask      = myRequestBitmap;
   myParamBlock.buffer.bufferSize = sizeof(MaximumLetterAttributes);
   myParamBlock.buffer.buffer    = myAttribPtr;
   myParamBlock.more             = false;

   /* call function to get the attributes */
   MSAMGetAttributes((MSAMParam *)&myParamBlock,true);
   myErr = DoWaitPBDone(&myParamBlock);
   if (myErr!=noErr)
      return myErr;

   /* save returned attributes to disk */
   if (myParamBlock.responseMask.indications) {
      myErr = DoWriteToFile(kMailIndicationsMask, myAttribPtr,
                            sizeof(MailIndications));
```

```
      if (myErr!=noErr)
         return myErr;
      myAttribPtr += sizeof(MailIndications);
   }

   if (myParamBlock.responseMask.msgType) {
      myErr = DoWriteToFile(kMailMsgTypeMask, myAttribPtr,
                            sizeof(OCECreatorType));
      if (myErr!=noErr)
         return myErr;
      myAttribPtr += sizeof(OCECreatorType);
   }

   if (myParamBlock.responseMask.letterID) {
      myErr = DoWriteToFile(kMailLetterIDMask, myAttribPtr,
                            sizeof(MailLetterID));
      if (myErr!=noErr)
         return myErr;
      myAttribPtr += sizeof(MailLetterID);
   }

   /*
      Test for presence of the send time stamp, nesting level, message
      family, reply ID, and conversation ID attributes. If present, write
      them to file.
   */

   if (myParamBlock.responseMask.subject) {
      myErr = DoWriteToFile(kMailSubjectMask, myAttribPtr, sizeof(RString));
      if (myErr!=noErr)
         return myErr;
      myAttribPtr += sizeof(RString);
   }
}
```

You can read information such as the message creator and message type from the message header of non-letter messages by calling the `MSAMGetMsgHeader` function.

Interpreting Creator and Type for Messages and Blocks

An outgoing message may have any message creator and any message type. Typically, an application that generates a message uses its own application signature as the message creator and its document type as the message type.

The message creator value `'lap2'` indicates that the AppleMail application created the message.

If the message type of an outgoing message is `kMailLtrMsgType`, the message is a letter that contains any or all of the following: data in standard interchange format, data in image format, or a regular enclosure.

Each block in an outgoing message has a block creator and block type. The AppleMail application sets the block creator to `kMailAppleMailCreator` for blocks that it creates. The block types that you may find in a letter are listed in Table 2-3 on page 2-18.

Reading Addresses

When you read the addresses associated with an outgoing message, you must get both the original and the resolved recipients for that message. That gives you complete addressing information for both display and routing purposes.

An ***original recipient*** can be a To, From, cc, or bcc recipient. These four types of original recipients are defined as follows:

- From: the sender of a message
- To: a primary recipient of a message
- cc: a secondary recipient receiving a copy of a letter
- bcc: a secondary recipient whose address does not appear on the letter as received by the To and cc recipients and other bcc recipients

An original recipient may be a group address (distribution list).

A ***resolved recipient*** is a recipient to which you are responsible for delivering the message. Usually, a resolved recipient is an individual address; sometimes it can be a group address.

Reading Original Recipients

To get a list of original recipients, you call the `MSAMGetRecipients` function. You need to get original recipients so that you can properly display them as From, To, cc, or bcc recipients in the message you send to an external messaging system. The function returns information about one type of original recipient. You specify the type of original recipient you want by setting the `attrID` field of the `MSAMGetRecipientsPB` parameter block appropriately. You can set the `attrID` field to any of the following constants:

Constant	Value	Recipient type
`kMailFromBit`	12	From
`kMailToBit`	13	To
`kMailCcBit`	14	cc
`kMailBccBit`	15	bcc

If you are reading a letter, you need to get each original recipient type so that when you translate the letter, it includes display information about all of the recipients. Display address information refers to an address that may not be usable for routing within a given messaging system but nevertheless shows that the letter went to the addressee.

(A bcc recipient is an exception, as it should be displayed only to the sender and the bcc recipient itself.)

If you are reading a non-letter message, the only original recipient types that apply are From and To. You may not need to get display information. If that is the case, do not call the `MSAMGetRecipients` function to retrieve the To recipients. You may still want to call it to get the From recipient. (You could also get the From recipient by calling the `MSAMGetMsgHeader` function.)

When a letter has a bcc recipient, you must make every attempt to conform to the following AOCE guidelines for bcc recipients: A bcc recipient must know that he or she is a bcc recipient. A To or a cc recipient must not see any bcc recipient. It is less desirable, but acceptable, for a bcc recipient to see other bcc recipients.

To support these guidelines, your MSAM may need to generate a separate copy of the letter for each bcc recipient for which it is responsible or employ other implementations that are less straightforward or more expensive than usual. As a last resort, if your MSAM cannot support AOCE guidelines, it must reject bcc recipients. In that case, it must still apply the guidelines to the letter—that is, no other recipient must know of the bcc recipients.

Reading Resolved Recipients

To get a list of resolved recipients, call the `MSAMGetRecipients` function and specify the `kMailResolvedList` constant in the `attrID` field of the `MSAMGetRecipientsPB` parameter block. You need to get a list of resolved recipients so that you know to which recipients you must send the message.

As you read the `MailResolvedRecipient` structures that the `MSAMGetRecipients` function places in your buffer, you must save the ordinal-position value for each resolved recipient. The first recipient's ordinal-position value is 1; the second recipient's ordinal-position value is 2; and so forth. The `MSAMnMarkRecipients` function requires you to provide the ordinal-position value to identify a recipient for whom you have completed delivery attempts. If you need to call `MSAMGetRecipients` more than once to get all of the resolved recipients, you must increment the ordinal-position value continuously so that each resolved recipient is associated with a unique ordinal-position value.

Personal MSAMs always find a one-to-one correspondence between their resolved recipients and their displayable (original) recipients because the `MSAMGetRecipients` function expands all group addresses into individual recipients before it returns recipient information to the personal MSAM.

Server MSAMs may find more recipients in the resolved list than in the displayable lists for this reason: the PowerShare mail server expands PowerShare group addresses into individual addresses for the resolved list, but the original recipient lists may have included PowerShare group addresses that were not expanded. The `MSAMGetRecipients` function does not expand external group addresses.

Server MSAMs may also find that there are recipients in the resolved list that are not exactly the same as the corresponding recipients in the original list. These have been resolved by the AOCE software to a more specific form.

The PowerShare mail server does not suppress duplicate external addresses. It does suppress duplicate addresses resulting from the expansion of a PowerShare group address. However, you are not guaranteed that the `MSAMGetRecipients` function will not return duplicate PowerShare addresses.

Listing 2-3 illustrates a dispatch routine that calls the `DoReadGenericAddress` function (shown in Listing 2-4 on page 2-55) to get a list of resolved recipients and lists of the original recipients that are appropriate to a letter or a non-letter message.

Listing 2-3 Getting resolved and original recipients

```
OSErr DoReadAddress(MailMsgRef myMsgRef)
{
   FSSpec   myTempFileSpec;
   OSErr    myErr;

   /* initialize the file specification */

   myErr = DoReadGenericAddress(&myTempFileSpec, myMsgRef,
                                kMailResolvedList);
   if (myErr!= noErr)
      return myErr;

   myErr = DoReadGenericAddress(&myTempFileSpec, myMsgRef, kMailFromBit);
   if (myErr!= noErr)
      return myErr;

   myErr = DoReadGenericAddress(&myTempFileSpec, myMsgRef, kMailToBit);
   if (myErr!= noErr)
      return myErr;

   if (myMsg->msgFamily == kMailFamily) { /* it's a letter */
      myErr = DoReadGenericAddress(&myTempFileSpec, myMsgRef, kMailCcBit);
      if (myErr!= noErr)
         return myErr;

      myErr = DoReadGenericAddress(&myTempFileSpec, myMsgRef, kMailBccBit);
      if (myErr!= noErr)
         return myErr;
   }

   return myErr;
}
```

The function DoReadGenericAddress shown in Listing 2-4 actually reads the addresses from an outgoing message and writes them to a disk file. The DoReadGenericAddress function takes three parameters: the file system specification of a temporary disk file to which it writes the addresses, the message reference number for a given message, and an attribute ID that identifies the type of address that the caller wants to retrieve from the message.

First DoReadGenericAddress allocates a buffer, pointed to by the addressBuffer field, that it uses to hold addresses returned by the MSAMGetRecipients function. It sets the size of the buffer to 1024 bytes. Your MSAM should determine the buffer size that is appropriate for your needs.

Next, DoReadGenericAddress determines if it is handling a request to get resolved or original recipients and sets the doingResolved Boolean variable accordingly. If it is handling resolved recipients, DoReadGenericAddress initializes its local variable ordinalPosition to 0. It uses ordinalPosition to save the ordinal position of each resolved recipient. It needs this information to mark a recipient when it has finished its efforts to deliver the letter to the recipient. The ordinal-position value must be unique for each recipient.

Then, DoReadGenericAddress fills in all but one of the fields of the local variable myParamBlock, which is an MSAMGetRecipientsPB parameter block. It sets the myParamBlock.mailMsgRef field to its message reference number parameter (myMailRef) to identify the message and sets the myParamBlock.attrID field to its attribute ID parameter (attrID) to indicate which type of address (To, From, cc, bcc, or resolved) it wants the MSAMGetRecipients function to return. Although the nextIndex and more fields are outputs of the MSAMGetRecipients function, DoReadGenericAddress sets them here to execute the for statement that follows and to initialize the myParamBlock.startIndex field properly the first time through the loop.

To accomplish its main work, DoReadGenericAddress uses two for loops, one nested inside the other. Note that the outer for statement contains only the logical expression controlling the iteration of the loop. The loop executes as long as the value of myParamBlock.more is true and no error has occurred. The MSAMGetRecipients function sets the more field to true when there are more addresses to return than it could fit into the caller's buffer.

The outer for loop sets the myParamBlock.startIndex field to the value of the myParamBlock.nextIndex field, which it previously set to 1. This tells the MSAMGetRecipients function that it should begin returning addresses starting with the first address of the specified type. Then DoReadGenericAddress calls MSAMGetRecipients asynchronously and polls for its completion.

If no error has occurred, DoReadGenericAddress initializes two variables used by the inner for loop. The MSAMGetRecipients function always puts at the beginning of your buffer the count of the number of addresses it placed in your buffer, followed by the addresses themselves. Therefore, DoReadGenericAddress sets recipientPtr to point into the address buffer at the byte where address information actually begins, skipping over the count. It next sets the variable numRecipients to the count of the

number of addresses in the buffer. Then, it executes the inner `for` loop to manipulate the addresses returned in the buffer.

The inner `for` loop extracts an address from the buffer and writes it to a disk file. It executes until all of the addresses have been extracted and written or until an error occurs. For convenience, `DoReadGenericAddress` defines two new types, `MailOriginalRecipientExt` and `MailResolvedRecipientExt`. Each consists of a `MailOriginalRecipient` or `MailResolvedRecipient` structure, respectively, followed by an `OCEPackedRecipient` structure. The new types enable `DoReadGenericAddress` to manipulate all of the relevant information associated with a particular address using a single structure.

If it is extracting resolved recipients, `DoReadGenericAddress` first increments the `ordinalPosition` local variable. Then it sets the pointer `resolvedPtr` to `recipientPtr`, which in turn points to the beginning of the first resolved address. The `DoReadGenericAddress` function writes the `MailResolvedRecipientExt` structure to a disk file, tagging it with its address type (attribute ID) and ordinal-position value for later identification. Once that is done, `DoReadGenericAddress` advances the `recipientPtr` pointer to the next address in the buffer. It moves `recipientPtr` past the `MailResolvedRecipient` structure, past the `dataLength` field in the `OCEPackedRecipient` structure, and then past the number of bytes specified in the `dataLength` field. If `recipientPtr` points to an odd byte address, `DoReadGenericAddress` increments it by 1 to point to an even byte boundary. At this point, the `for` loop is ready to execute again.

Because of differences in the sizes of the applicable structures, the `for` loop has separate but parallel logic to extract and write resolved and original recipients.

The logic of `DoReadGenericAddress` assumes that after it writes the addresses to disk, the MSAM translates them from AOCE address format into the format of the destination messaging system.

Listing 2-4 Reading addresses from an outgoing message

```
OSErr DoReadGenericAddress(FSSpec *myTempFileSpec, MailMsgRef myMailRef,
                           MailAttributeID attrID)
{
   typedef struct MailOriginalRecipientExt {
      MailOriginalRecipient   prefix;
      OCEPackedRecipient      packedRecip;
   } MailOriginalRecipientExt;

   typedef struct MailResolvedRecipientExt {
      MailResolvedRecipient   prefix;
      OCEPackedRecipient      packedRecip;
   } MailResolvedRecipientExt;
```

```
OSErr                       myErr;
MSAMGetRecipientsPB         myParamBlock;
short                       count, numRecipients, ordinalPosition;
MailOriginalRecipientExt    *origPtr;
MailResolvedRecipientExt    *resolvedPtr;
Ptr                         addressBuffer, recipientPtr;
Boolean                     doingResolved;

addressBuffer = NewPtr(1024L);
if (MemError()!= noErr)
   return MemError();

if (attrID == kMailResolvedList) {
   doingResolved     = true;
   ordinalPosition   = 0;
else
   doingResolved = false;

myParamBlock.ioCompletion        = (ProcPtr)DoMSAMCompletion;
myParamBlock.mailMsgRef          = myMailRef;
myParamBlock.attrID              = attrID;
myParamBlock.buffer.buffer       = addressBuffer;
myParamBlock.buffer.bufferSize   = 1024L;
myParamBlock.more                = true; /* to get into "for" loop */
myParamBlock.nextIndex           = 1;

myErr = noErr;
for ( ; myParamBlock.more == true && myErr == noErr; ) {
   myParamBlock.startIndex = myParamBlock.nextIndex;
   MSAMGetRecipients((MSAMParam *)&myParamBlock,true);
   myErr = DoWaitPBDone(&myParamBlock);
   if (myErr != noErr) {
      DisposPtr(addressBuffer);
      return myErr;
   }  /* end if */
   recipientPtr   = addressBuffer + sizeof(short);
   numRecipients  = (MailReply *) addressBuffer->tupleCount;
   for (count = 0; count < numRecipients && myErr == noErr;
         count++) {
      if (doingResolved) {
         resolvedPtr  = (MailResolvedRecipientExt *)recipientPtr;
         ordinalPosition++;
         myErr = WriteRecipient(myTempFileSpec, attrID, resolvedPtr,
                                ordinalPosition);
```

```
            recipientPtr += (sizeof(MailResolvedRecipient) + sizeof(short)
                              + resolvedPtr->packedRecip.dataLength);
            if ((unsigned long)recipientPtr % 2)/*pad to even boundary */
               recipientPtr++;
         }  /* end if */
         else {
            origPtr      = (MailOriginalRecipientExt *)recipientPtr;
            myErr = WriteRecipient(myTempFileSpec, attrID, origPtr, 0);
            recipientPtr += (sizeof(MailOriginalRecipient) + sizeof(short)
                              + origPtr->packedRecip.dataLength);
            if ((unsigned long)recipientPtr % 2)/*pad to even boundary */
               recipientPtr++;
         }  /* end else */
      }  /* end inner for loop */
   }  /* end outer for loop */

   DisposPtr(addressBuffer);
   return myErr;
}
```

Reading Letter Content

You read a letter's content block by calling the `MSAMGetContent` function. A content block consists of a series of data segments. A segment contains data in any of these formats: plain text, styled text, pictures, sound, and QuickTime movies. You select which types of segment you want to read by setting the `segmentMask` field in the function's parameter block appropriately.

To read the segments sequentially, set the `segmentID` field to 0. The `MSAMGetContent` function returns data from the first segment of a type that you requested in your segment mask. Continue resetting the `segmentID` field to 0 on subsequent calls to the `MSAMGetContent` function to read the segments of interest sequentially.

To access the segments in any order you choose, set the `segmentID` field to a given segment's segment ID. You can obtain the segment ID for each segment in a letter's content block by scanning the segments without actually reading in any data. To do this, set the `segmentMask` and `segmentID` fields to 0 before calling the `MSAMGetContent` function. This tells the function that you do not want it to return data for any segment type and that you want it to return information about the segments starting with the first segment in the block. Save the values of the `segmentType`, `segmentLength`, and `segmentID` fields that the function returns. Reset the `segmentID` field to 0 and call the function again to get information about the next segment in the block. Continue saving the values of the `segmentType`, `segmentLength`, and `segmentID` fields, resetting the `segmentID` field to 0, and calling the function. The function provides information about the next segment in the content block. When it returns information about the last segment in the content block, the function returns `true` in the `endOfContent` field.

At this point, you know the order of the segments in the block, the type of data each contains, the number of bytes in the segment, and the segment IDs. You can then read the data in the segments in any order you choose. Set the `segmentMask` field to indicate the types of segments from which you want to retrieve data. The types of segment data you request depends on the capabilities of your messaging system. For instance, if your messaging system understands only plain text data, there is no point in reading segments that contain QuickTime movie data.

The function `DoReadLetterContent` in Listing 2-5 reads a letter's content block. It allocates buffer space for the segment data. In the `MSAMGetContentPB` parameter block, it sets the segment mask to request data from segments containing plain text, pictures, and sound. Then it repeatedly calls the `MSAMGetContent` function until the function returns `true` in the `endOfContent` field, always resetting the segment ID to 0 to proceed sequentially through the blocks. If `MSAMGetContent` completes successfully, `DoReadLetterContent` writes the segment data to a file. Later, it can read this file and build its message in the format acceptable to its external messaging system.

Listing 2-5 Reading a letter's content block

```
#define  kMaxBufferSize    32767L

OSErr DoReadLetterContent(FSSpec *myTempFileSpec, MailMsgRef myMailRef)
{
   MSAMGetContentPB     myParamBlock;
   Ptr                  dataBuffer;
   OSErr                myErr;
   Boolean              startOfBlock;
   unsigned short       blockIndex;


   /* allocate data buffer */
   dataBuffer = NewPtr(kMaxBufferSize);
   if (MemError() != noErr)
      return MemError();

   /* fill in parameter block */
   myParamBlock.ioCompletion        = (ProcPtr)DoMSAMCompletion;
   myParamBlock.mailMsgRef          = myMailRef;
   myParamBlock.buffer.buffer       = dataBuffer;
   myParamBlock.buffer.bufferSize   = kMaxBufferSize;
   myParamBlock.segmentMask         = kMailTextSegmentMask |
                            kMailPictSegmentMask | kMailSoundSegmentMask;
   myParamBlock.textScrap           = nil;
```

```
   /* read letter content */
   startOfBlock   = true;
   blockIndex     = 0;
   do {
      myParamBlock.segmentID = 0;
      MSAMGetContent((MSAMParam *)&myParamBlock,true);
      myErr = WaitPBDone(&myParamBlock);
      if ((myErr == noErr) && (myParamBlock.buffer.dataSize > 0)) {
         if (startOfBlock) {
            DoWriteContentToFile(myTempFileSpec, myParamBlock.segmentType,
                                 myParamBlock.buffer.buffer,
                                 myParamBlock.buffer.dataSize, blockIndex);
            startOfBlock = false;
         }
         else
            DoAppendContentToFile(myTempFileSpec, myParamBlock.segmentType,
                                 myParamBlock.buffer.buffer
                                 myParamBlock.buffer.dataSize, blockIndex);
         if (myParamBlock.endOfSegment == true) {
            startOfBlock = true;
            blockIndex++;
         }
      }
   } while ((myErr == noErr) && (myParamBlock.endOfContent == false));

   DisposPtrChk(dataBuffer);

   return myErr;
}
```

Reading a Nested Message

A message can have other messages nested within it. If you are reading a letter, you can determine if the letter contains nested letters by calling the `MSAMGetAttributes` function and requesting the `nestingLevel` attribute. A nesting level of 0 means there are no nested letters; a nesting level of 1 means there is one nested letter, and so forth. If you are reading a non-letter message, you can determine if it contains a nested message by calling the `MSAMEnumerateBlocks` function and looking for a block of type `kIPMEnclosedMsgType`. Such a block contains a complete message. That nested message may in turn contain a message block of type `kIPMEnclosedMsgType` that contains a complete message, and so on.

To open a nested message, you call the `MSAMOpenNested` function, which returns a reference number to the nested message. To read the nested message, you pass this nested message reference number to functions. An MSAM can call `MSAMOpenNested` repeatedly to open a hierarchy of nested messages.

You can close a nested message explicitly by calling the `MSAMClose` function or you can close it implicitly when you close the parent message.

Note

A letter can have only one nested letter per nesting level, although each nested letter can itself contain a nested letter, and so forth. A non-letter message may actually have more than one nested message per nesting level. The IPM Manager API allows applications to create such messages. However, the MSAM API restricts you to reading one nested message per nesting level. You can read only the first occurrence of a nested message in a sequence of message blocks. ◆

Marking Recipients

Once you have read a message from the outgoing queue, translated it into the format understood by your external messaging system, and transmitted it, you can mark one or more recipients. Marking a recipient indicates that you have completed your efforts to deliver the message to that recipient. You mark a recipient by calling the `MSAMnMarkRecipients` function.

Marking a recipient does not indicate that you have successfully delivered the message, but only that you are finished with your efforts to deliver it to that recipient.

You can use the `MSAMnMarkRecipients` function to help you keep track of your delivery status for a message. The function clears the `responsible` flag in the `MailResolvedRecipient` structure for the recipients you specify. Thus, if you later call the `MSAMGetRecipients` function to get the resolved recipients for the message, the `responsible` flag indicates those recipients you have already processed.

You identify a recipient that you want to mark by its ordinal position in the buffer returned by the `MSAMGetRecipients` function. That is, when you call the `MSAMGetRecipients` function to get your resolved recipients, it places recipient information in your buffer, and you must save the ordinal-position value of each resolved recipient as you retrieve the recipient information from the buffer. The first recipient's ordinal-position value is 1; the second recipient's ordinal-position value is 2; and so forth. It is this value that you provide to the `MSAMnMarkRecipients` function to identify the recipient. If you use the recipient's absolute index, contained in a `MailResolvedRecipient` structure, the `MSAMnMarkRecipients` function does not work correctly.

After you mark all of the recipients for a given message, the function sets the `done` field in the `MSAMEnumerateOutQReply` structure to `true`. If you later call the `MSAMEnumerate` function to check the messages in your outgoing queue, you can determine if you have finished processing a given message by checking the `done` field.

You can call the `MSAMnMarkRecipients` function as many times as necessary for a given message, specifying one or more recipients each time as you complete your delivery efforts for those recipients.

Generating a Report

When you have completed your delivery attempts for an outgoing message, you may need to generate a report to the sender. An MSAM determines whether it must create a report for an outgoing message by reading information in the message header. An MSAM should create a report about an outgoing message only in response to the sender's request.

If the message is a letter, an MSAM calls the `MSAMGetAttributes` function to read the `MailIndications` structure. In the `MailIndications` structure, the `kMailNonReceiptReportsBit` bit and the `kMailReceiptReportsBit` bit, if set, indicate that the letter's sender requested non-delivery and delivery indications, respectively.

If the message is not a letter, an MSAM calls the `MSAMGetMsgHeader` function with the constant `kIPMFixedInfo` as the value of the `selector` field. The `IPMFixedHdrInfo` structure returned by `MSAMGetMsgHeader` contains the `notification` field, which contains the `kIPMNonDeliveryNotificationBit` bit and the `kIPMDeliveryNotificationBit` bit. These bits, if set, indicate that the sender of the message requested non-delivery and delivery indications, respectively. Test these bits to determine if you need to create a report.

If a sender asks for delivery indications, non-delivery indications, or both, an MSAM must provide information on the outcome of delivery attempts (a delivery or non-delivery indication) for every recipient for which the MSAM is responsible. It is important that an MSAM provide delivery information on all of the MSAM's recipients whenever a sender requests any type of delivery information because an MSAM report does not go directly to the report requestor. Instead, the report goes to an AOCE agent that uses the MSAM report information to prepare an IPM report according to the requestor's specifications. If an MSAM fails to provide delivery information on all of its recipients, the requestor may receive inaccurate IPM reports.

An MSAM should ignore the bit fields having to do with including a copy of the original message in the report. If necessary, a copy of the original is added by the AOCE agent.

To create a report, an MSAM must

1. call the `MSAMCreateReport` function
2. call the `MSAMPutRecipientReport` function to add delivery and non-delivery indications for recipients for which it was responsible
3. call the `MSAMSubmit` function to deliver its finished report

An MSAM must have certain information about a message in order to create a report about the message. The `MSAMCreateReport` function requires the letter or message ID of the message to which the report applies and the address of the sender. You obtain this information from either the `MSAMGetAttributes` and `MSAMGetRecipients` functions (for a letter) or the `MSAMGetMsgHeader` function (for a non-letter message). The `MSAMPutRecipientReport` function requires the recipient index to identify which recipient is being reported upon. You obtain this information from the `MSAMGetRecipients` function.

Depending on how the external messaging system works, an MSAM may save this information in its own data store or include it with the message. If, for example, more than one MSAM connects to the same external messaging system, and the system might acknowledge receiving the message to any of those MSAMs, an MSAM should include the information with the message. This enables the external messaging system to extract the information from the message and then include the information with the acknowledgment of the message. As a result, any MSAM that receives the acknowledgment has the information necessary to create a report for that message. You decide how to make sure that the information required to create a report is available, given the characteristics of the external messaging system to which your MSAM connects.

Your MSAM and its external messaging system define what constitutes successful or failed delivery for outgoing messages.

Writing Incoming Messages

This section describes how you create and submit an incoming letter for delivery to its AOCE recipients. It assumes you have already initialized your MSAM. Each subsection addresses a specific task, such as

- creating a message summary for an incoming letter (for personal MSAMs only)
- creating a letter
- creating a non-letter message
- writing letter attributes
- writing addresses
- writing letter content
- submitting a letter for delivery
- receiving a report

The differences between writing letters and writing non-letter messages are noted in the sections that address the specific tasks. For convenience, Table 2-10 lists the tasks you perform while handling incoming messages and the functions you use to accomplish each task for a letter and a non-letter message.

The order in which functions are listed in Table 2-10 corresponds to the sequence in which you would call the functions to process an incoming message. A personal MSAM first creates a message summary if it is dealing with a letter. Then all MSAMs create the message itself and begin adding information to it. First, you write header information consisting of message attributes, such as the priority of the message, and address (recipient) information. Next, you write the substance of the message—for a letter, its content block, other blocks it may contain, and enclosures; for a non-letter message, its blocks. You can include an entire message within another message by defining its beginning and end with the `MSAMBeginNested` and `MSAMEndNested` functions and

Table 2-10 Incoming tasks and functions

Task	Letters	Non-letter messages
Create a messaging summary (personal MSAMs only)	`PMSAMCreateMsgSummary`	Not applicable
Create a message	`MSAMCreate`	`MSAMCreate`
Write header information	`MSAMPutAttribute` `MSAMPutRecipient`	`MSAMPutMsgHeader` `MSAMPutRecipient`
Write letter content	`MSAMPutContent`	Not applicable
Write an enclosure	`MSAMPutEnclosure`	Not applicable
Write a block	`MSAMPutBlock`	`MSAMPutBlock`
Write a nested letter	`MSAMBeginNested` `MSAMEndNested`	`MSAMBeginNested` `MSAMEndNested`
Submit a message	`MSAMSubmit`	`MSAMSubmit`
Delete a message (personal MSAMs only)	`MSAMDelete`	Not applicable
Set message status (personal MSAMs only)	`PMSAMSetStatus`	Not applicable
Enumerate a queue (personal MSAMs only)	`MSAMEnumerate`	Not applicable

calling the appropriate functions to write the nested message's header information, blocks, enclosures, and so forth. When you have finished writing the message, you submit it to the AOCE system for delivery to its recipients.

A personal MSAM may also delete a letter, or both the letter and the letter's message summary, from an incoming queue. For example, the MSAM may delete a letter (but not the message summary) if it no longer wants the letter to be cached locally. If the personal MSAM is mirroring the letter's status on the external messaging system, it can delete the letter and message summary when the letter is removed from the external messaging system.

A personal MSAM may also set the status of a letter and enumerate an incoming queue. Setting the status of a letter is a task that the MSAM performs at several points while it is processing the letter. Enumerating an incoming queue is a task it may do in response to receiving a `kMailEPPCInQUpdate` high-level event.

You should call the functions that handle incoming messages asynchronously so that you can receive and process an AOCE high-level event at any time.

The sample code in Listing 2-6 through Listing 2-15 illustrates one way a personal MSAM can write a letter to an incoming queue. Most of the sample code and the text also apply to a server MSAM. The text notes differences between the operation of personal and server MSAMs where applicable.

Most of these listings contain code fragments from the `DoIncomingLetter` function, but only Listing 2-6 on page 2-67 shows the `DoIncomingLetter` function definition and its local variables.

Choosing Creator and Type for Messages and Blocks

When you create an incoming message, you set the message creator to indicate the application that should open the message. If you set the message creator for a letter to `'lap2'`, the signature of the AppleMail application, the AppleMail application opens the letter when the user double-clicks the letter's icon. If the letter contains a content enclosure, you can set the message creator to the signature of the application that created the content enclosure. In this case, if the user has that application, that application will open the letter.

The message type `kMailLtrMsgType` designates an AOCE letter that contains data in standard interchange format or image format, or a regular enclosure. When you create an incoming letter, you should use this message type when the letter contains data in standard interchange format or image format, or when it contains a regular enclosure. If the letter also contains a content enclosure or a private block, and you set the message creator to the signature of the application that created the enclosure or private block, then you can use a message type that you define that is consistent with the message creator.

When you create a non-letter message, you typically use an application-defined message creator and message type.

Each block in an incoming message has a block creator and block type. When you create blocks such as header, content, enclosure, and report blocks by calling the appropriate MSAM function, the function sets the block creator to `kMailAppleMailCreator` and the block type to the correct predefined type. (Letter block types are listed in Table 2-3 on page 2-18.)

When you call the `MSAMPutBlock` function to add a block to an incoming message, you set the block creator and block type to values that you select. If you are writing a block of a predefined type such as an image block or a private block, be sure to set the block type to `kMailImageBodyType` or `kMailMSAMType`, respectively.

Creating a Letter's Message Summary

A personal MSAM must create a message summary for an incoming letter before creating the letter itself. Server MSAMs do not create message summaries at any time, and personal MSAMs do not create message summaries for non-letter messages. The need to create a message summary is related to the mode of operation in the personal MSAM. See the section "MSAM Modes of Operation" beginning on page 2-12 for information on this topic.

The function `DoIncomingLetter` shown in Listing 2-6 on page 2-67 illustrates how you can create a message summary for an incoming letter. It assumes that you previously read the letter from an external messaging system, translated it into AOCE data formats,

and saved it to disk. (Note that this method is just one way an MSAM can handle incoming letters.)

The `DoIncomingLetter` function first allocates the buffer `dataBuffer` that it uses to hold a variety of data throughout the function's execution. Then it initializes all of the fields of the message summary structure to 0 prior to setting the fields that a personal MSAM should set. At the top level of the message summary structure, `DoIncomingLetter` sets only the `version` field. You always set it to the constant `kMailMsgSummaryVersion`.

You set the bits in the attribute mask that correspond to the attributes that are present in the letter. In the `attrMask` field of the `masterData` substructure, `DoIncomingLetter` sets the bits for the send timestamp, indications, the sender of the letter, the subject of the letter, the message type, and the message family. Each external messaging system may differ in the attribute information it routinely provides. In the sample code, the external messaging system always provides a timestamp and does not provide a reply ID. For this reason, the corresponding bits in the attribute mask in the message summary are set and not set accordingly.

Once you have set the bits in the attribute mask, you write the attributes to the message summary. At a minimum, you must write the message type, send timestamp, sender, and subject attributes to the message summary. The `DoIncomingLetter` function first writes the send timestamp to the message summary by calling its `DoGetTimeStamp` routine. Next, it calls its `DoGetLetterLength` utility routine to get the approximate size of the letter.

In the `coreData` substructure, `DoIncomingLetter` explicitly provides a value for all of the fields except `agentInfo` and `letterFlags`. (The `DoIncomingLetter` function implicitly set the `letterFlags` field to 0 when it initialized the entire message summary structure to 0.) In the `letterIndications` field, it sets those bits that indicate the letter has normal priority and that it has a content block. This technique assumes that the incoming letter has no priority setting, so `DoIncomingLetter` supplies a default value here. (The `DoIncomingLetter` function also supplies a default value for content if the letter has no content. See Listing 2-11 on page 2-78.)

The `DoIncomingLetter` function sets the message type to the constant `kMailLtrMsgType` to indicate a standard AOCE letter. It sets the message creator to `kLetterCreator`, a constant for `'lap2'`, the signature of the AppleMail application. As a result, when a user double-clicks the letter, the Finder launches the AppleMail application to open the letter. Usually, an MSAM does not set a letter's creator to its own signature because the MSAM cannot open the letter and allow the user to view and edit it. However, if your MSAM is associated with a particular letter application, you should use that application's signature so that the application will launch when the user opens the letter.

The `DoIncomingLetter` function sets the message family to `kMailFamily`, indicating that the letter falls into the general class of mail messages. Next, it sets the `messageSize` field to the value returned by the `DoGetLetterLength` utility routine. The Finder uses this value when a user chooses the Get Info command from the File menu.

The `sender` and `subject` fields in the message summary deserve special attention. Each is declared as an `RString32` structure in the `MailCoreData` structure in the message summary. However, those declarations only serve to allocate space and indicate the relative order of the sender and subject data. They do not represent the actual data layout. You should treat these two fields as a common buffer containing variable-length sender and subject data. The correct order of information in the common buffer is an `RString32` structure containing the sender information (character set, data length, and sender data), padded to an even byte boundary if necessary, and followed immediately by an `RString32` structure containing the subject information. (You should also pad the subject information to an even byte boundary if necessary.) Thus, sender information always starts at a fixed place whereas subject information does not. Neither subject nor sender information may exceed `kRString32Size` bytes although either, of course, may be smaller.

The `DoIncomingLetter` function illustrates one way to write the sender and subject information to a message summary. The `DoIncomingLetter` function calls its `DoReadFromFile` utility routine to read a `PackedDSSpec` structure containing the sender's address information from the letter stored on disk. (The `DoReadFromFile` routine reads a file in which an incoming letter is stored and returns in a buffer the requested letter component and the number of bytes it placed in the buffer.) If the read operation succeeds, `DoIncomingLetter` unpacks the packed address and calls its `DoCopyFitRString` utility routine. The `DoCopyFitRString` routine copies the displayable string that identifies the sender from the `recordName` field of the unpacked address into the `sender` field of the message summary, truncating it if it is longer than `kRString32Size` bytes.

Next, `DoIncomingLetter` reads into its local variable `subject` an `RString` structure containing the subject from the stored letter. Every AOCE letter must have a subject. If the read operation fails, `DoIncomingLetter` converts a constant C string containing a default value for the subject into an `RString` and writes it to its local variable `subject`. Finally, it calls its `DoCopyFitRString` routine to copy its local variable `subject` into the message summary, truncating it if it is longer than `kRString32Size` bytes. (The `DoIncomingLetter` function copies the subject into its local variable `subject` instead of directly into the message summary because it uses the local variable when adding the subject attribute to the letter header. See Listing 2-8 on page 2-72.)

Now that both the subject and sender information are in a common buffer in the message summary, `DoIncomingLetter` adjusts the byte position at which the subject information begins. The subject information must start immediately after the sender information. `DoIncomingLetter` calculates the total length of the sender `RString`, including the fields for length and character set. If the total is an odd number, it adds 1 to get an even word boundary, then calls the `BlockMove` routine to move the subject information immediately after the end of the sender information.

IMPORTANT

Because the `sender` and `subject` fields form one common buffer into which the information is packed, using the `subject` field to access the subject information does not produce the desired result. You must compute the beginning of the subject information in the common buffer. ▲

At this point, the `DoIncomingLetter` function has filled in the relevant fields of the message summary. Next, it sets up the fields of the parameter block for the `PMSAMCreateMsgSummary` function. One of the parameters to `DoIncomingLetter` is a `MySlotSpec` structure, a data type defined by the personal MSAM that contains information about a slot. The personal MSAM of which `DoIncomingLetter` is a part previously stored the incoming queue reference that it obtained from the `PMSAMOpenQueues` function in the `MySlotSpec` structure. The `DoIncomingLetter` function uses that incoming queue reference to fill in the `queueRef` field of the `MSAMCreate` parameter block. Next, it sets the `msgSummary` field of the parameter block to the address of the message summary structure it has just initialized. Although `DoIncomingLetter` does not do it, you can add up to `kMailMaxPMSAMMsgSummaryData` bytes of private data in the buffer structure pointed to by the `buffer` field of the `PMSAMCreateMsgSummary` parameter block. It is a convenient way for you to store additional information related to a specific letter. Then `DoIncomingLetter` calls the `PMSAMCreateMsgSummary` function, which returns a sequence number for the letter. The `DoIncomingLetter` function must use this sequence number when it calls the `MSAMCreate` function to create the letter itself.

Listing 2-6 Creating a message summary

```
OSErr DoIncomingLetter(FSSpec *myTempFileSpec, MySlotSpec *slotSpec)
{
   OSErr             myErr;
   MSAMParam         myParamBlock;
   MSAMMsgSummary    myMsgSum;
   Ptr               dataBuffer;
   unsigned long     bufferLen;
   unsigned long     contentLength;
   RString           subject;
   RecordID          entitySpecifier;
   OCERecipient      fromAddress;
   MailMsgRef        letterRef;
   long              letterSeqNum;
   char              defaultText[256];
   unsigned char     *subjectOffset;

   #define  kLetterCreator    'lap2'   /* signature of AppleMail app */
   #define  kDefaultSubject   "<no subject>"
```

```
#define  kDefaultBody      "<no message>"
#define  kMaxBufferSize    32767L
/* constants to identify components of stored letter on disk */
#define  kFromType         '2FRM'
#define  kToType           '2MTO'
#define  kCCType           '2MCC'
#define  kBCCType          '2BCC'
#define  kTextContent      '2TXT'
#define  kPictContent      '2PIC'
#define  kSoundContent     '2SND'
#define  kContentSectionType  '2RTY'
#define  kSubjectType      '2SUB'


/* allocate buffer for reading from disk */
bufferLen   = kMaxBufferSize;
dataBuffer  = NewPtr(bufferLen);
if (MemError() != noErr)
   return MemError();

/* initialize the message summary structure to 0 */
DoClearBuffer(&myMsgSum,sizeof(MSAMMsgSummary));

/* set the version and attribute mask fields */
myMsgSum.version                            = kMailMsgSummaryVersion;
myMsgSum.masterData.attrMask.sendTimeStamp   = true;
myMsgSum.masterData.attrMask.indications     = true;
myMsgSum.masterData.attrMask.from            = true;
myMsgSum.masterData.attrMask.subject         = true;
myMsgSum.masterData.attrMask.msgType         = true;
myMsgSum.masterData.attrMask.msgFamily       = true;

/* get the timestamp and write it to message summary */
DoGetTimeStamp(myTempFileSpec,&myMsgSum.coreData.sendTime);

/* get length of stored letter data in bytes */
contentLength = kMaxBufferSize;
contentLength = DoGetLetterLength(myTempFileSpec);

/* set other core data fields */
myMsgSum.coreData.letterIndications.priority    = kIPMNormalPriority;
myMsgSum.coreData.letterIndications.hasContent  = true;
myMsgSum.coreData.letterIndications.hasStandardContent= true;
```

```
   myMsgSum.coreData.messageType.msgType          = kMailLtrMsgType;
   myMsgSum.coreData.messageType.msgCreator       = kLetterCreator;
   myMsgSum.coreData.messageFamily                = kMailFamily;
   myMsgSum.coreData.messageSize                  = contentLength;
   myMsgSum.coreData.addressedToMe                = kAddressedAs_TO;

/* get sender name from stored letter and write it to message summary */
   bufferLen = kMaxBufferSize;
   myErr = DoReadFromFile(myTempFileSpec, kFromType, dataBuffer,
                          &bufferLen);
   if (myErr != noErr) {
      DisposPtr(dataBuffer);
      return myErr;
   }
   OCEUnpackDSSpec((PackedDSSpec*)dataBuffer, &fromAddress,
                   &entitySpecifier);
   DoCopyFitRString(entitySpecifier.local.recordName,
                (RStringPtr)&myMsgSum.coreData.sender, kRString32Size);


/* get subject from stored letter and write it to message summary */
   bufferLen = kMaxBufferSize;
   myErr = DoReadFromFile(myTempFileSpec, kSubjectType, &subject,
                          &bufferLen);
   if (myErr != noErr)
      OCECToRString(kDefaultSubject, smRoman, &subject, kRStringMaxBytes);
   DoCopyFitRString(&subject, (RStringPtr)&myMsgSum.coreData.subject,
                    kRString32Size);

/* calculate subject offset and move subject flush with sender */
   subjectOffset = ((unsigned char *)&myMsgSum.coreData.sender) +
                    myMsgSum.coreData.sender.dataLength + sizeof(long);
   if ((unsigned long)subjectOffset % 2)
      subjectOffset++;
   BlockMove(&myMsgSum.coreData.subject, subjectOffset,
             myMsgSum.coreData.subject.dataLength + sizeof(long));

/*
   All required fields have been set. Create the message summary. Save the
   letter's sequence number.
*/
   myParamBlock.header.ioCompletion             = (ProcPtr)DoMSAMCompletion;
   myParamBlock.pmsamCreateMsgSummary.inQueueRef   = slotSpec->inQueue;
```

```
myParamBlock.pmsamCreateMsgSummary.msgSummary   = &myMsgSum;
myParamBlock.pmsamCreateMsgSummary.buffer       = nil;
PMSAMCreateMsgSummary(&myParamBlock,true);
myErr = DoWaitPBDone(&myParamBlock);
if (myErr != noErr) {
   DisposPtr(dataBuffer);
   return myErr;
}
letterSeqNum = myParamBlock.pmsamCreateMsgSummary.seqNum;
```

Creating a Letter

After creating a message summary, a personal MSAM may write the letter associated with the message summary to the incoming queue immediately or at a later time. The choice of methods should depend on the speed of the link connecting your personal MSAM to its external messaging system. If the link is fast, you can download the letter on demand—that is, when the user opens it. If the link is slow, you should cache the letter locally so that there is no untimely delay when the user opens it. The function `DoIncomingLetter` writes the letter immediately. Listing 2-7 is a code fragment from `DoIncomingLetter` that shows how you create a letter.

The `DoIncomingLetter` function sets up the fields of the parameter block for the `MSAMCreate` function. It checks whether the letter has a blind copy recipient and sets the `bccRecipients` field accordingly. It uses the incoming queue reference originally obtained from the `PMSAMOpenQueues` function to fill in the `queueRef` field of the parameter block. Then `DoIncomingLetter` sets the `asLetter` field to `true` to indicate that the message it is creating is a letter. Because it is creating a letter, it must set the `msgType.format` field to `kIPMOSFormatType`. This setting indicates that the rest of the `IPMMsgType` structure contained in the `msgType.format` field consists of an `OCECreatorType` structure. Then `DoIncomingLetter` sets the letter's creator and type to the same values it used when it created the letter's message summary. It sets the `seqNum` field to the sequence number it obtained from the `PMSAMCreateMsgSummary` function.

Once `DoIncomingLetter` has finished initializing the parameter block, it calls the `MSAMCreate` function. The function returns a reference to the new letter, which `DoIncomingLetter` saves. The `DoIncomingLetter` function must provide the reference to all subsequent functions that add various components to the letter.

Listing 2-7 Creating a letter

```
/* check for bcc recipients */
bufferLen = kMaxBufferSize;
myErr = DoReadFromFile(myTempFileSpec, kBCCType, dataBuffer, &bufferLen);
myParamBlock.msamCreate.bccRecipients  = (myErr == noErr);
```

```
/* fill in the rest of the parameter block and create the letter */
myParamBlock.header.ioCompletion       = (ProcPtr)DoMSAMCompletion;
myParamBlock.msamCreate.queueRef       = slotSpec->inQueue;
myParamBlock.msamCreate.asLetter       = true;
myParamBlock.msamCreate.msgType.format = kIPMOSFormatType;
myParamBlock.msamCreate.msgType.theType.msgOSType.msgCreator =
                                          kLetterCreator;
myParamBlock.msamCreate.msgType.theType.msgOSType.msgType =
                                          kMailLtrMsgType;
myParamBlock.msamCreate.seqNum         = letterSeqNum;
myParamBlock.msamCreate.tunnelForm     = false;
MSAMCreate(&myParamBlock, true);
myErr = DoWaitPBDone(&myParamBlock);
if (myErr != noErr) {
   DisposPtr(dataBuffer);
   return myErr;
}
letterRef = myParamBlock.msamCreate.newRef;
```

A server MSAM does basically the same things to create a letter, with the following differences. A server MSAM uses the queue reference that it obtained from the `SMSAMStartup` function to fill in the `queueRef` field. Because server MSAMs do not create message summaries, there is no need to ascertain that the values provided to the `MSAMCreate` function for the creator and type exactly match those in the message summary. A server MSAM does not supply a value in the `seqNum` field of the `MSAMCreate` parameter block.

Creating a Non-Letter Message

When you create a non-letter message instead of a letter, the following differences apply for both personal and server MSAMs:

- You must set the `myParamBlock.msamCreate.asLetter` field to `false`.
- You can set the `myParamBlock.msamCreate.msgType.format` field to either `kIPMOSFormatType` (which specifies that the message creator and message type information is formatted as type `OCECreatorType`) or `kIPMStringFormatType` (which specifies that the message creator and message type information is formatted as type `Str32`). Typically, you use type `OCECreatorType`; type `Str32` is included for compatibility with the Program-to-Program Communications (PPC) Toolbox.
- You may set the `myParamBlock.msamCreate.refCon` field to a private value. The `MSAMCreate` function stores that value in the message header. A recipient can retrieve the value with the `MSAMGetMsgHeader` function.
- You do not supply a value in the `myParamBlock.msamCreate.bccRecipients` field.

In addition, a personal MSAM does not supply a value in the `myParamBlock.msamCreate.seqNum` field.

Writing Letter Attributes

Once you have created a letter, you add the component parts to the letter. To add information to a letter's header, you use the `MSAMPutAttribute` function. Listing 2-8, a code fragment from the `DoIncomingLetter` function, shows how you add attributes to a letter header.

The `MSAMPutAttribute` function allows you to add one attribute each time you call it. The `DoIncomingLetter` function adds the send timestamp, indications, message family, and subject attributes to the letter's header by copying the values it previously stored in the letter's message summary. Each time it calls the `MSAMPutAttribute` function, `DoIncomingLetter` sets the `mailMsgRef` field to indicate the letter to which it wants to add the attribute. It sets the `attrID` field to a constant that indicates the type of attribute it wants to add. Then it specifies the buffer in which the attribute data is located, specifies the buffer size, and calls the `MSAMPutAttribute` function to add the attribute to the letter header. Note that when it writes the subject, `DoIncomingLetter` does not use the C function `sizeof` to get the size of the subject attribute because that would return the size of an `RString` structure. Instead, it computes the exact size of the subject string in the buffer by using the actual length of the subject, which is specified in the `subject.dataLength` field, and then adding 4 bytes for the `dataLength` and `charSet` fields of the `RString` structure. If the number of bytes turns out to be odd, it adds 1 to make an even length.

The `DoIncomingLetter` function does not add the letter creator and type to the letter header. That information was already added when `DoIncomingLetter` called the `MSAMCreate` function.

Once the parameter block is initialized, `DoIncomingLetter` calls the `MSAMPutAttribute` function. If the function returns an error, `DoIncomingLetter` calls its `DoCancelOnSubmit` function, which disposes of the data buffer, calls the `MSAMSubmit` function to delete the unfinished letter, and calls the `MSAMDelete` function to delete the message summary.

Listing 2-8 Adding attributes to a letter header

```
/* add the time */
myParamBlock.msamPutAttribute.mailMsgRef        = letterRef;
myParamBlock.msamPutAttribute.attrID            = kMailSendTimeStampBit;
myParamBlock.msamPutAttribute.buffer.buffer     =
                                      (Ptr)&myMsgSum.coreData.sendTime;
myParamBlock.msamPutAttribute.buffer.bufferSize = sizeof(MailTime);
MSAMPutAttribute(&myParamBlock, true);
myErr = DoWaitPBDone(&myParamBlock);
if (myErr != noErr) {
   DoCancelOnSubmit(letterRef, letterSeqNum, slotSpec->inQueue,
                    dataBuffer);
   return myErr;
}
```

```
/* add the indications */
myParamBlock.msamPutAttribute.mailMsgRef       = letterRef;
myParamBlock.msamPutAttribute.attrID           = kMailIndicationsBit;
myParamBlock.msamPutAttribute.buffer.buffer    =
                            (Ptr)&myMsgSum.coreData.letterIndications;
myParamBlock.msamPutAttribute.buffer.bufferSize = sizeof(MailIndications);
MSAMPutAttribute(&myParamBlock, true);
/*
   Call DoWaitPBDone and check for error. Then use the same logic used
   to add the time and indications to add the message family.
*/

/* add the subject */
myParamBlock.msamPutAttribute.mailMsgRef       = letterRef;
myParamBlock.msamPutAttribute.attrID           = kMailSubjectBit;
myParamBlock.msamPutAttribute.buffer.buffer    = (Ptr)&subject;
myParamBlock.msamPutAttribute.buffer.bufferSize = subject.dataLength + 4;
if ((myParamBlock.msamPutAttribute.buffer.bufferSize % 2) != 0)
   myParamBlock.msamPutAttribute.buffer.bufferSize++;
MSAMPutAttribute(&myParamBlock, true);
/* call DoWaitPBDone and check for error */
```

A server MSAM does not have a message summary from which to copy attribute values, so it would extract the attribute values from the incoming letter itself.

Note
The `MSAMPutAttribute` function does not apply to non-letter messages. In dealing with an incoming non-letter message, both personal and server MSAMs can add attributes to the message header by calling the `MSAMPutMsgHeader` function. ◆

Writing Addresses

Although the different types of recipients—From, To, cc, and bcc—are letter attributes, you do not add them to a letter using the `MSAMPutAttribute` function. Instead, you use the `MSAMPutRecipient` function. Each time you call the `MSAMPutRecipient` function, you can add one recipient to a letter. This function requires you to add all of the recipients of one type before adding any recipient of another type. The code fragment from the `DoIncomingLetter` function shown in Listing 2-9 demonstrates how you can add recipients to a letter.

The `DoIncomingLetter` function calls its `DoAddTheRecipients` function four times, once for each type of recipient, to actually add the recipient information to the letter. It passes several parameters to `DoAddTheRecipients`:

- the reference number of the letter to which it wants to add a recipient
- a pointer to the file specification of the temporary file containing the translated incoming letter

- a constant that identifies the disk file component for a given type of recipient
- the type of recipient to add (an attribute ID)
- a pointer to its buffer
- the size of the buffer

If `DoAddTheRecipients` returns an error for any type of recipient, `DoIncomingLetter` terminates writing the letter.

Listing 2-9 Adding recipients to a letter

```
/*
   Add the recipients. Check for error after calling DoAddTheRecipients
   for each recipient type.(Shown only the first time in the following
   code.)
*/
myErr = DoAddTheRecipients(letterRef, myTempFileSpec, kFromType,
                           kMailFromBit, dataBuffer, kMaxBufferSize);
if (myErr != noErr) {
   DoCancelOnSubmit(letterRef, letterSeqNum, slotSpec->inQueue,
                    dataBuffer);
   return myErr;
}

myErr = DoAddTheRecipients(letterRef, myTempFileSpec, kToType, kMailToBit,
                           dataBuffer, kMaxBufferSize);

myErr = DoAddTheRecipients(letterRef, myTempFileSpec, kCcType, kMailCcBit,
                           dataBuffer, kMaxBufferSize);

myErr = DoAddTheRecipients(letterRef, myTempFileSpec, kBccType,
                           kMailBccBit, dataBuffer, kMaxBufferSize);
```

The `DoAddTheRecipients` function is shown in Listing 2-10. It is a utility routine that can add any type of recipient to a given letter. It assumes that the MSAM has previously written the letter's recipient information to a file in the form of a `PackedDSSpec` structure. For a given type of recipient, `DoAddTheRecipients` reads one recipient at a time, and places the information in a buffer. Then it unpacks the `PackedDSSpec` structure and fills in the fields of the parameter block for the `MSAMPutRecipient` function.

The `DoAddTheRecipients` function sets the `mailMsgRef` and `attrID` fields to the values it was passed by `DoIncomingLetter` for the letter's reference number and the recipient type attribute ID, respectively. It sets the `recipient` field to the unpacked `DSSpec` structure it got by calling the `OCEUnpackDSSpec` routine. Then it sets the `responsible` field to `false`.

A personal MSAM always sets the `responsible` field of the parameter block for `MSAMPutRecipient` to `false` when it is adding a recipient to a letter. For a non-letter message, however, it should set the `responsible` field to `false` only when the recipient address is not local to the computer on which the personal MSAM is running. Setting the `responsible` field to `true` for a non-letter message indicates that you want the AOCE system to be responsible for delivering the message to its destination on the local computer.

A server MSAM should set the `responsible` field to `true` to indicate that the AOCE system should deliver the message to the recipient. This applies to both letter and non-letter messages.

Finally, `DoAddTheRecipients` calls the `MSAMPutRecipient` function. The `DoAddTheRecipients` function repeats this cycle until either the `MSAMPutRecipient` function returns an error or there are no more recipients of a given type for the letter.

Listing 2-10 Adding a specific type of recipient

```
OSErr DoAddTheRecipients(MailMsgRef mailRef, FSSpec *myTempFileSpec,
                         OSType recipType, MailAttributeID attrID,
                         Ptr dataBuffer, unsigned long bufferLen)

{
   OSErr          myErr;
   Boolean        moreRecipients = true;
   unsigned long  gotLength;
   OCERecipient   recipient;
   RecordID       entitySpecifier;
   MSAMParam      myParamBlock;


   do {
      gotLength = bufferLen;
      myErr = DoReadFromFile(myTempFileSpec, recipType, dataBuffer,
                             &gotLength);
      if (myErr == noErr && gotLength > 0) {

         /* unpack a recipient, initialize the parameter block,
            add the recipient */
         OCEUnpackDSSpec((PackedDSSpec*)dataBuffer, &recipient,
                         &entitySpecifier);
         myParamBlock.msamPutRecipient.ioCompletion   =
                                             (ProcPtr)DoMSAMCompletion;
         myParamBlock.msamPutRecipient.mailMsgRef  = mailRef;
         myParamBlock.msamPutRecipient.attrID      = attrID;
```

```
            myParamBlock.msamPutRecipient.recipient   = &recipient;
            myParamBlock.msamPutRecipient.responsible = false;
            MSAMPutRecipient(&myParamBlock, true);
            myErr = DoWaitPBDone(&myParamBlock);
         }
         else {
            moreRecipients    = false;
            myErr             = noErr;
         }

      } while (myErr == noErr && moreRecipients);

      return myErr;
}
```

Writing Letter Content

A letter's content block consists of a series of one or more segments, each containing data of one of the following types: plain text, styled text, pictures, sounds, and QuickTime movies. To add a content block to an incoming letter, you call the `MSAMPutContent` function.

You provide the function with a buffer containing data of a given type and tell it what type of data is in the buffer. The first time you call the `MSAMPutContent` function, set the `append` field to `false` to tell the function to begin a new segment. On subsequent calls to the function, you set the `append` field to `true` or `false`, depending on whether you want your data placed in a new segment or appended to the current one.

When you add a text segment, you must specify values for the `startNewScript` and `script` fields. The value of the `startNewScript` field (`true` or `false`) tells the `MSAMPutContent` function whether the data in your buffer uses a different character set than that of text data you previously wrote. You set the `script` field to a code that indicates the character set of your data. (See *Inside Macintosh: Text* for a list of script codes.)

When you add a styled text segment, you provide the style information in a style scrap structure (`StScrpRec` structure). You should allocate the `StScrpRec` structure dynamically because it is a very large structure. See the `MSAMPutContent` function description on page 2-186 for more information on adding styled text.

You must add all of a letter's content sequentially. For instance, you cannot call `MSAMPutContent` to add some of the content, call `MSAMPutBlock` to add a private block, and then call `MSAMPutContent` again to add the remainder of the content. Once you call `MSAMPutContent`, calling any other function in the MSAM API terminates the content block for the letter. If you call the `MSAMPutContent` function again for the same letter, it returns the `kMailInvalidOrder` result code. The `MSAMPutContent` function adds the segments to the letter in the order you provide them.

The `DoWriteLetterContent` function in Listing 2-11 shows one way to add content to an incoming letter. It assumes the MSAM has previously stored a letter from its external messaging system in a disk file. The file is composed of a series of sections corresponding to different components of the letter. The content component of the stored letter consists of a series of sections, similar to the segments in a letter's content block, each of which contains a single type of data.

The `DoWriteLetterContent` function starts by initializing the fields of the `MSAMPutContent` function's parameter block that won't change regardless of what it reads from its file. It sets the `mailMsgRef` field to the letter's reference number. It sets the `textScrap` field to `nil` because it does not handle styled text. Because this MSAM handles just one character set, `DoWriteLetterContent` sets the `script` field to `smRoman` and never changes this setting. It sets the `append` field to `false` because it intends that each block of data that it previously stored on disk be written to a separate segment in the letter's content block.

The `DoWriteLetterContent` function initializes its local variable `contentType` to indicate that it wants to read the content section of its stored letter. It sets the local variable `contentWritten` to `false` because it has not yet written a segment to the incoming letter.

Then `DoWriteLetterContent` reads sequentially through the content sections of the stored letter. It repeatedly calls the `DoReadFromFile` utility routine to read a buffer of data from the file. The `DoReadFromFile` function returns one content section from the file each time it is called. The buffer is large enough to hold any content section that the MSAM previously stored. After reading each section, `DoWriteLetterContent` determines the type of data in the section and sets the `segmentType` field accordingly. Because this MSAM handles only plain text, picture, or sound data, the content sections can contain only these types of data. If `DoReadFromFile` returns plain text data, `DoWriteLetterContent` sets the `startNewScript` field to `true`. This tells the `MSAMPutContent` function to examine the `script` field to discover the character set of the text in the buffer. Typically, you set this field to `true` when you first add a plain text segment and thereafter whenever the character set of the text changes (which does not apply to this MSAM) or you've called `MSAMPutContent` to add some other type of segment. Last, `DoWriteLetterContent` sets the `bufferSize` field to the number of bytes it read from its disk file and calls the `MSAMPutContent` function to write the data to the letter's content block. If the `MSAMPutContent` function returns successfully, `DoWriteLetterContent` sets the local variable `contentWritten` to `true`. The `DoWriteLetterContent` function continues to read from its file and write segments to the letter's content block until it has read all the content sections in the file or it encounters an error.

When `DoWriteLetterContent` has finished reading the content sections, it tests the local variable `contentWritten`. If it failed to write any data successfully, `DoWriteLetterContent` copies a default string into its buffer and calls the `MSAMPutContent` function. It must do this to provide some content since it set the `hasContent` bit in the `indications` attribute in the letter's header. (See Listing 2-6 on page 2-67.)

Listing 2-11 Writing letter content

```
OSErr DoWriteLetterContent(FSSpec *myTempFileSpec, MailMsgRef myMailRef,
                           Ptr dataBuffer)
{
   unsigned long  bufferLen;
   OSType         contentType;
   Boolean        contentWritten;
   MSAMParam      myParamBlock;
   OSErr          myErr,myErr2;

   myParamBlock.header.ioCompletion          = (ProcPtr)MSAMCompletion;
   myParamBlock.msamPutContent.mailMsgRef    = myMailRef;
   myParamBlock.msamPutContent.textScrap     = nil;
   myParamBlock.msamPutContent.buffer.buffer = dataBuffer;
   myParamBlock.msamPutContent.script        = smRoman;
   myParamBlock.msamPutContent.append        = false;

   contentType       = kContentSectionType;
   contentWritten    = false;

   do {        /* for each content section in the temp file */
      bufferLen = kMaxBufferSize;
      myErr = DoReadFromFile(myTempFileSpec, contentType, dataBuffer,
                             &bufferLen);
      switch (contentType) {  /* determine segment type */
         case kTextContent:
            myParamBlock.msamPutContent.segmentType = kMailTextSegmentType;
            myParamBlock.msamPutContent.startNewScript = true;
            break;
         case kPictContent:
            myParamBlock.msamPutContent.segmentType = kMailPictSegmentType;
            break;
         case kSoundContent:
            myParamBlock.msamPutContent.segmentType = kMailSoundSegmentType;
            break;
      } /* endswitch */
      myParamBlock.msamPutContent.buffer.bufferSize= bufferLen;
      if (myErr == noErr) {
         MSAMPutContent(&myParamBlock,true);
         myErr2 = WaitPBDone(&myParamBlock);
         if (myErr2 != noErr)
            return myErr2;
         contentWritten = true; /* don't need default content */
```

```
      }  /* endif */
   } while (myErr != noErr);

   if (myErr == kEndOfContentSections)
      myErr = noErr;

   /* if no content written, write default content */
   if (contentWritten == false) {
      strcpy(dataBuffer,kDefaultBody);
      myParamBlock.msamPutContent.segmentType = kMailTextSegmentType;
      myParamBlock.msamPutContent.buffer.bufferSize = strlen(kDefaultBody);
      MSAMPutContent(&myParamBlock,true);
      myErr = WaitPBDone(&myParamBlock);
   }

   return myErr;
}
```

You call the `MSAMPutContent` function to add content to letters only. You do not call it to write data to a non-letter message.

Submitting a Message

After composing a message, an MSAM calls the `MSAMSubmit` function to submit the message to the AOCE system for delivery. A message must be complete before you submit it because, when the `MSAMSubmit` function completes execution, the message's reference number is invalid and you cannot change the message in any way.

Listing 2-12 is a code fragment from the `DoIncomingLetter` function that shows how you can submit a letter for delivery. The `DoIncomingLetter` function sets the `mailMsgRef` field to the letter's reference number and the `submitFlag` field to `true` to indicate that the letter is ready for delivery. If you set the `submitFlag` field to `false`, the function deletes the letter. Then `DoIncomingLetter` calls the `MSAMSubmit` function.

If `MSAMSubmit` returns an error, `DoIncomingLetter` calls the `MSAMDelete` function to delete the message summary associated with the letter. The `DoIncomingLetter` function sets the `queueRef` field to the reference value that identifies the incoming queue in which the message summary is located. (It originally obtained this value from the `PMSAMOpenQueues` function.) Then it sets the `seqNum` field to the sequence number that identifies the message summary. Last, `DoIncomingLetter` sets the `msgOnly` field to `false`. This tells `MSAMDelete` to delete the letter and its message summary. In this case, there is no letter to delete. The `MSAMDelete` function deletes the message summary and returns the result code `noErr`.

Listing 2-12 Submitting a letter

```
/* submit the letter */
myParamBlock.msamSubmit.mailMsgRef = letterRef;
myParamBlock.msamSubmit.submitFlag = true;
myErr = MSAMSubmit(&myParamBlock);
if (myErr != noErr) {                        /* delete message summary */
   myParamBlock.msamDelete.queueRef = slotSpec->inQueue;
   myParamBlock.msamDelete.seqNum   = msgSeqNum;
   myParamBlock.msamDelete.msgOnly  = false;
   myParamBlock.msamDelete.result   = noErr;
   MSAMDelete(&myParamBlock, true);
   DoWaitPBDone(&myParamBlock);
}
DisposPtr(dataBuffer);

return myErr;
```

If `DoIncomingLetter` had been dealing with a non-letter message, it would not need to delete a message summary, because a personal MSAM only creates a message summary for a letter. A server MSAM, of course, does not need to delete a message summary because it never creates one.

Because it normally has continuous access to the PowerShare mail server, a server MSAM should translate incoming messages immediately and submit them to the PowerShare mail server. If the PowerShare mail server quits, the server MSAM should either stop accepting incoming messages or store the incoming messages until the PowerShare mail server is available again.

Receiving a Report

An MSAM can receive reports about incoming messages. Server MSAMs can receive reports on both letters and non-letter messages. Personal MSAMs can receive reports on non-letter messages only.

To request a report on a non-letter message, an MSAM should set the appropriate bits in the `deliveryNotification` field when it calls the `MSAMPutMsgHeader` function. You set the bits by using the `kIPMDeliveryNotificationMask` or `kIPMNonDeliveryNotificationMask` masks to request delivery and non-delivery indications.

To request a report on a letter, a server MSAM should set the `receiptReports` bit, the `nonReceiptReports` bit, or both in the letter's `MailIndications` attribute.

Because personal MSAMs do not receive reports on letters, the IPM Manager ignores the setting of the `receiptReports` and `nonReceiptReports` bits in a letter's `MailIndications` attribute for any letter submitted by a personal MSAM. Instead, the result code of the `MSAMSubmit` function tells a personal MSAM if the letter delivery attempt was successful or not.

The report that an MSAM receives never includes a copy of the original message. Thus, the IPM Manager ignores the bits in a letter's indications attribute and a non-letter message's header that have to do with enclosing a copy of the original with the report.

An MSAM can identify a report from the IPM Manager in its outgoing queue because all such reports have a message creator of `kIPMSignature` and a message type of `kIPMReportNotify`.

An MSAM reads a report by calling the `MSAMOpen`, `MSAMGetMsgHeader`, and `MSAMGetBlock` functions. Reports consist of a recipient report block (type `kMailReportType`) and possibly a private data block (type `kMailMSAMType`). The recipient report block contains a report header and information about some number of recipients. (See the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces* for a description of the report header `IPMReportBlockHeader` and the recipient report information structure `OCERecipientReport`.) If an MSAM added a private data block to a message, the IPM Manager includes a copy of that block in the report.

A report may contain information on one or more AOCE recipients. The IPM Manager attempts to report as quickly as possible on each recipient. If there is some difficulty in reporting, it sends a report on the recipients about which it has information and sends another report about the remaining recipients at a later time. Therefore, if a message that the MSAM put into an AOCE system has several recipients, the MSAM may get several reports. If the MSAM plans to forward that information to its external messaging system, it may want to consolidate the information from the reports before forwarding it.

Note
The AOCE software defines successful delivery to mean that the message was placed in the recipient's incoming queue. It does not imply that the message was actually opened or read. ◆

Deleting a Message

A personal MSAM should not delete messages from its outgoing queues. Messages should stay in an outgoing queue so that the user can look at them. An exception to this rule occurs when a user wants to delete a letter rather than send it. In that case, the IPM Manager sends the personal MSAM a `kMailEPPCDeleteOutQMsg` event, and the MSAM should delete the letter. A server MSAM does delete messages from its outgoing queue.

A personal MSAM can delete letters from an incoming queue. It can delete only a letter or both a letter and the associated message summary. For example, the MSAM may want to delete a letter, but not the message summary, when it decides the letter no longer needs to be cached locally. If the MSAM is trying to mirror the letter's status on its external messaging system, it can delete the letter and the message summary when the letter is removed from the external messaging system.

Note
The IPM Manager may also delete a letter from a personal MSAM's incoming queue in response to a user action. In that case, it sets the `msgDeleted` flag in the letter's message summary and sends the `kMailEPPCInQUpdate` event. ◆

The `MSAMDelete` function removes a message from the queue that you specify. You identify the message by its sequence number, which you obtain from the `MSAMEnumerate` function. Once you have deleted a message, it is no longer available to you on the Macintosh computer on which your MSAM is running. (The message may still exist on the external messaging system.)

Translating Addresses

One of an MSAM's primary tasks is translating address information from AOCE format to the format of its external messaging system and vice versa. Within AOCE software, an address is defined by an `OCERecipient` structure, a complex structure that contains other structures and elemental fields. It is described on page 2-106. Figure 2-13 on page 2-28 illustrates the fields in an `OCERecipient` structure and their relationship to each other. Table 2-4 on page 2-29 lists what each field should contain for a non-AOCE address. Table 2-5 on page 2-30 lists the contents of each field when the `OCERecipient` structure contains an AOCE address. If you are already familiar with the information in Figure 2-13, Table 2-4, and Table 2-5, you'll find the listings and descriptions in the sections "Translating From an AOCE Address" and "Translating to an AOCE Address" easier to understand.

Note that an `OCERecipient` structure is identical to a `DSSpec` structure.

Within this chapter and the MSAM API, an address is often referred to as an *xxx recipient*, where *xxx* specifies a type of recipient—To, From, cc, or bcc.

A non-letter message contains only From and To recipients. A letter may contain any type of recipients.

An address can become known to an AOCE system by any of the following methods:

- the user provides the address information by means of an address template (see the chapter "Service Access Module Setup" in this book for an explanation of address templates)
- the address is read from an incoming message
- the user types in the address when using a mailer (this works only if the extension value portion of the address is formatted as a single `RString`; see the chapter "Standard Mail Package" in *Inside Macintosh: AOCE Application Interfaces* for an explanation of the mailer and type-in addressing)
- the address exists in a catalog and can be retrieved by the user or an application

The MSAM whose code is shown in the sections that follow is a personal MSAM that connects to an SMTP messaging system. The address format understood by the SMTP messaging system is a string of this form: *username@systemlocation*. The information presented applies to server MSAMs as well.

Translating From an AOCE Address

Prior to transmitting a letter to its external messaging system, an MSAM must convert the address information from AOCE format (an `OCERecipient` structure) to the format understood by its external messaging system.

The function `DoBuildSMTPAddressInfo` in Listing 2-13 provides an example of building a non-AOCE address from an `OCERecipient` structure. The `DoBuildSMTPAddressInfo` function first allocates a buffer pointed to by `addressBuf`. This address buffer will eventually hold all of the SMTP address information for a given letter except the bcc recipients, which are stored in a separate buffer. The `DoBuildSMTPAddressInfo` function sets the first byte in the address buffer to 0 to indicate an empty string.

When it is launched, this MSAM creates and maintains a `MySlotSpec` structure for each mail slot for which it is responsible. This privately defined structure contains all the information relevant to a individual slot. To build the From address, the `DoBuildSMTPAddressInfo` function begins by copying the user name from the `MySlotSpec` structure for the slot it is processing into the local variable `fromAddr`. Then the function appends to the user name the @ character and the SMTP server name, which it also copies from the `MySlotSpec` structure. Once it has finished building the string holding the actual From address, `DoBuildSMTPAddressInfo` builds a second string in the address buffer that includes formatting information. First, it copies the constant `kMyFromHeader` into `addressBuf` to label the address. The constant's value is `"From: "`. Next, it appends the From address in `fromAddr` to the contents of the address buffer. Finally, it appends a carriage return. At this point, the contents of the address buffer look like this:

`From: username@systemLocation`(CR)0

Next, `DoBuildSMTPAddressInfo` adds the To addresses. To the address buffer, it adds the string `"To: "` to label the address. It initializes the `hasRecipient` Boolean variable to `false` to indicate that at this point it has found no To recipients. Then it repeats the following procedure until it encounters an error:

- Read a To address from a temporary file. The MSAM created this file when it read the letter from AOCE. If there are no more To addresses, it will get an error here.
- If the read succeeded
  - call the `DoAOCEToSMTPAddress` function (see Listing 2-14 on page 2-87), which converts an AOCE address into an SMTP address
  - append the SMTP address and a comma to the contents of the address buffer
  - set the `hasRecipient` Boolean to `true`

At this point, `DoBuildSMTPAddressInfo` completes the formatting. If it added any To addresses to the address buffer, it overwrites the last comma with the string terminator 0 and then appends a carriage return. The contents of the address buffer now look like this:

`From: username@systemLocation`(CR)`To: recipient1@location, recipient2@location,...,recipientN@location`(CR)0

If it has not added any To addresses to the address buffer, it positions the string terminator 0 immediately before the `"To: "` label, in effect erasing it.

The `DoBuildSMTPAddressInfo` function processes a letter that has no To recipient for two reasons. First, AOCE software considers valid a letter whose header has at least one To, cc, or bcc recipient. Therefore, it is possible for an MSAM to get a letter from its AOCE system that has no To recipient. Second, as you will see in Listing 2-14 on page 2-87, this MSAM translates only SMTP addresses. It is possible that all of the To recipients for a given letter are non-SMTP addresses, but that one or more of the cc or bcc addresses are SMTP addresses. This topic is discussed in more detail in the explanation of Listing 2-14.

The `DoBuildSMTPAddressInfo` function adds the cc addresses to the address buffer in exactly the same manner as it added the To address. At this point, the address buffer contains a string that includes the From, To, and cc addresses, formatted with commas and carriage returns, and terminated by a `NULL` character.

For bcc addresses, `DoBuildSMTPAddressInfo` uses the same procedure but a separate buffer, `bccBuf`. Typically, an SMTP messaging system does not display a bcc address even to a bcc recipient. Therefore, `DoBuildSMTPAddressInfo` places any bcc addresses in a separate buffer so they can be handled separately. In code not shown in Listing 2-13, the `DoBuildSMTPAddressInfo` function uses the information in the address buffer for both routing and display purposes, but it uses the address information in the bcc buffer for routing only.

When `DoBuildSMTPAddressInfo` has finished building its two address buffers, it adds them to the letter.

Listing 2-13 Building SMTP addresses

```
OSErr DoBuildSMTPAddressInfo(FSSpec *myTempFileSpec, MySlotSpec *slotSpec)
{
   #define  kMyMaxAddrBufSize    4096     /* this MSAM's limit on address
                                             info */
   #define  kMyFromHeader        "From: "
   #define  kMyToHeader          "To: "
   #define  kMyCCHeader          "Cc: "
   #define  kMyBCCHeader         "Bcc: "
   #define  kMyAddressDelimiter  ", "
   #define  kMyCRStr             "\r"

   OSErr          myErr;
   char           tmpString[256];
   char           bccBuf[256];
   char           fromAddr[256];
   char           *addressBuf;
   unsigned long  tmpLen;
```

```
char           packedRecip[kMaxRecipSize];
Boolean        hasRecipient;

/* allocate memory to hold addresses in external form */
addressBuf = NewPtr(kMyMaxAddrBufSize);
if (MemError() != noErr) {
   return (MemError();
}
addressBuf[0] = 0;

/* build 'from' address */
strcpy(fromAddr, slotSpec->dirIdentity.userName);
strcat(fromAddr, "@");
strcat(fromAddr, slotSpec->specInfo.smtpServer);
strcpy(addressBuf, kMyFromHeader);
strcat(addressBuf, fromAddr);
strcat(addressBuf, kMyCRStr);

/* build 'To' address */
hasRecipient = false;
strcat(addressBuf, kMyToHeader);
for (myErr = noErr; myErr == noErr; ) {
   tmpLen = kMaxRecipSize;
   myErr = DoReadFromFile(myTempFileSpec, kToType, (Ptr)packedRecip,
                          &tmpLen);
   if (myErr == noErr) {
      if (DoAOCEToSMTPAddress(
                        (OCEPackedRecipient *)packedRecip, tmpString)) {
         strcat(addressBuf, tmpString);
         strcat(addressBuf, kMyAddressDelimiter);
         hasRecipient = true;
      }
   }
}
if (hasRecipient) {
   addressBuf[strlen(addressBuf) - strlen(kMyAddressDelimiter)] = 0;
   strcat(addressBuf, kMyCRStr);
}
else {
   addressBuf[strlen(addressBuf) - strlen(kMyToHeader)] = 0;
}

/* not shown here -- build 'cc' address just like 'To' address */
```

```
   /* build 'bcc' address just like 'To' address but in separate buffer */
   hasRecipient = false;
   strcpy(bccBuf,kMyBCCHeader);
   for (myErr=noErr; myErr==noErr; ) {
      tmpLen = kMaxRecipSize;
      myErr = DoReadFromFile(myTempFileSpec,kBCCType, (Ptr)packedRecip,
                             &tmpLen);
      if (myErr==noErr) {
         if (DoAOCEToSMTPAddress(
                             (OCEPackedRecipient *)packedRecip,tmpString)) {
            strcat(bccBuf,tmpString);
            strcat(bccBuf,kMyAddressDelimiter);
            hasRecipient = true;
         }
      }
   }
   if (hasRecipient) {
      bccBuf[strlen(bccBuf)-strlen(kMyAddressDelimiter)] = 0;
      strcat(bccBuf,kMyCRStr);
   }

   /* not shown here -- add address information to the letter */

   DisposPtr(addressBuf);
   return noErr;
}
```

The `DoAOCEToSMTPAddress` function in Listing 2-14 converts an SMTP address contained in an `OCEPackedRecipient` structure into string format. It returns `true` when it produces an SMTP address from an `OCEPackedRecipient` structure.

The `DoAOCEToSMTPAddress` function calls the `OCEUnpackDSSpec` AOCE utility routine to unpack the packed recipient information pointed to by its `packedRecip` parameter. If the extension type of the unpacked address specifies an SMTP address, it calls the `BlockMove` function to copy the value from the `extensionValue` field into the `RString` structure `recipRString`, converts the `RString` in `recipRString` into a C string, and stores the C string in the buffer pointed to by its `unixRecip` parameter. Then it returns `true`. If the extension type specifies some other type of address, the `DoAOCEToSMTPAddress` function makes no effort to translate the address and simply returns `false`.

A user can send a single letter to recipients in different types of messaging systems; thus, a single AOCE letter header may contain addresses with different extension types. This creates a potential problem for an MSAM, which is illustrated in the following example. The SMTP messaging system to which our sample MSAM is connected understands

Listing 2-14 Converting from AOCE to SMTP address

```
Boolean DoAOCEToSMTPAddress(OCEPackedRecipient *packedRecip,
                                    char *unixRecip)
{
   #define  kMySMTPAddrType'SMTP'

   OCERecipient   recip;
   RecordID       entitySpecifier;
   OSType         recipType;
   RString        recipRString;


   OCEUnpackDSSpec((PackedDSSpec*)packedRecip, &recip, &entitySpecifier);
   recipType = recip.extensionType;
   switch (recipType) {
      case kMySMTPAddrType:
         BlockMove(recip.extensionValue, &recipRString, recip.extensionSize);
DoRToCString(&recipRString, unixRecip);
         break;
      default:                   /* if not SMTP address, don't convert it */
         return false;
         break;
   }
   return true;
}
```

only SMTP addresses. When the messaging system receives a letter, it tries to route the letter to all of the addresses in the letter header. If it cannot do this, it generates an error reply to the sender. Suppose an AOCE user sends a letter to a fax address and sends a copy to a recipient with an SMTP address. Our sample MSAM is responsible for this SMTP address and must deliver the letter to the SMTP recipient. How should the MSAM handle the fax address? It cannot add the fax address as the To recipient because the SMTP messaging system will complain. Yet, it should provide the SMTP recipient with a letter that shows that the letter's primary recipient was a fax address.

The solution to this dilemma is up to the MSAM and its messaging system. For instance, the MSAM can copy the displayable strings from the `recordName` and `recordType` fields of an address into a display area in the letter header. A messaging system does not interpret information in the header's display area. If no such display area exists, the MSAM can append the displayable strings to the body of the letter and note that the letter was also sent to that address.

An MSAM can add an actual address for which it is not responsible instead of the displayable strings from the `recordName` and `recordType` fields of the address. To do this, it must know the address format specified by a given extension type and how an

address of that type is stored in an `OCERecipient` structure. Knowing this, the MSAM can translate the extension value into an actual address. (Apple does not define the syntax and semantics for non-AOCE address extension types. MSAM developers must work together to define agreed-upon extension types, and the associated address syntax and semantics.)

Suppose, for example, an AppleLink MSAM knows how an SMTP address is stored in an `OCERecipient` structure. If an AOCE user sends a letter to an AppleLink address and to an SMTP address, the AppleLink MSAM can translate the SMTP address to its proper SMTP form and add it to the letter header as a display address.

Remember that an MSAM only delivers a letter to those recipients for which it is responsible. All other recipient information with the letter is for display purposes only, regardless of whether the other recipient information is included in actual address format or as displayable strings, and regardless of where the information is stored (a display area in the letter header or the body of the letter).

Note
Given that an MSAM routes a letter only to those recipients for which it is responsible, a recipient on the MSAM's messaging system cannot necessarily reply to all other recipients. An MSAM must consider what to do when a recipient wants to reply to addresses that the MSAM cannot reach. Regardless of how it handles this situation, the MSAM should avoid sending the AOCE user a reply that looks as if it went to all recipients of the original message if in fact it did not. ◆

Although an MSAM is limited by the characteristics of the messaging system to which it is connected, it should always attempt to represent all recipients of an outgoing letter that it translates and transmits.

Translating to an AOCE Address

When an MSAM receives a message from its external messaging system, it must translate the addresses associated with the message before it can deliver the message to an AOCE system.

The function `DoConvertToAOCEAddress` in Listing 2-15 on page 2-90 provides an example of building an AOCE `OCERecipient` address structure from a non-AOCE address. The `DoConvertToAOCEAddress` function takes an address from a letter it received from its SMTP system and puts that address into AOCE format. The `DoConvertToAOCEAddress` function calls several AOCE utility routines to facilitate the process of constructing an AOCE address; the utility routines are described in the chapter "AOCE Utilities" in *Inside Macintosh: AOCE Application Interfaces*.

Listing 2-15 picks up at the point where `DoConvertToAOCEAddress` begins assembling the pieces of an `OCERecipient` structure. The `DoConvertToAOCEAddress` function begins by constructing the record ID part of the `OCERecipient`. A record ID, in turn, consists of a local record ID and record location information. It makes an `RLI` structure that contains the record location information by calling the AOCE utility routine `OCENewRLI` and providing it with an `RLI` structure's component parts: a catalog name, a discriminator, a dNode number, and a path. The `OCENewRLI` function returns

the RLI structure. The MSAM retrieves the catalog name from the private slot specification structure (type MySlotSpec) that the MSAM builds when it is launched. Because dNode numbers and paths are not used with non-AOCE addresses, DoConvertToAOCEAddress passes OCENewRLI a null dNode number and a nil pointer to a path. After OCENewRLI returns the RLI structure, DoConvertToAOCEAddress calls the AOCE utility routine OCEValidRLI to check its validity.

Next, DoConvertToAOCEAddress calls the OCEPackRLI utility routine to convert the RLI structure into packed form and calls the OCEValidPackedRLI utility routine to check the validity of the packed form.

Having prepared the record location information, DoConvertToAOCEAddress next prepares the local record ID, which consists of a creation ID, a record name, and a record type. A creation ID is not used in a non-AOCE address, so DoConvertToAOCEAddress calls the OCESetCreationIDtoNull utility routine to set the CreationID structure to 0. The buffer pointed to by the local variable realName contains a displayable form of the sender or receiver's name in C string format. The DoConvertToAOCEAddress function converts the C string into an RString and stores the RString in the local variable recordName. It tells the OCECToRString utility routine what character set the string uses and how many bytes, at maximum, it should place in the data portion of the RString, which in this example is the maximum number of bytes. Then DoConvertToAOCEAddress calls the OCECToRString utility routine again to get an RString that contains the sender or receiver's type. In this example, the type is always set to the constant kUserRecTypeBody, indicating a user.

At this point, DoConvertToAOCEAddress calls the OCENewLocalRecordID utility routine to build a local record ID from the creation ID, record name, and record type. The DoConvertToAOCEAddress function then calls the OCENewRecordID utility routine to build a record ID from its packed RLI and local record ID.

At last, DoConvertToAOCEAddress is ready to build the OCERecipient itself. It sets the entitySpecifier field to point to the record ID it has just constructed. Then it sets the extension fields. It specifies its extension type in the extensionType field. The buffer pointed to by the local variable startAddr contains the SMTP address in C string format. The DoConvertToAOCEAddress function converts the C string into an RString and stores the RString in the local variable xtnValueRString. (The DoConvertToAOCEAddress function converts the extension value from C string to RString format so that the mailer can correctly display the SMTP address to the user.) Then, DoConvertToAOCEAddress sets the extensionSize field to the number of bytes in the body field of xtnValueRString plus 4 more to account for the dataLength and charSet fields in an RString structure. This produces a count of the total number of bytes in xtnValueRString. Last, DoConvertToAOCEAddress sets the extensionValue field to point to xtnValueRString.

Before writing the address to a disk file, DoConvertToAOCEAddress converts the address into packed form. It calls the OCEPackedDSSpecSize utility routine, passing it the unpacked structure. In response, OCEPackedDSSpecSize returns the size of the packed structure into which the unpacked structure could be converted. Then DoConvertToAOCEAddress calls the OCEPackDSSpec utility routine and passes the

size value to it. Finally, `DoConvertToAOCEAddress` writes the packed structure to a disk file.

Listing 2-15 Building an `OCERecipient` structure

```
OSErr DoConvertToAOCEAddress(FSSpec *myTempFileSpec, MySlotSpec *slotSpec)
{
   #define kMySMTPAddrType    'SMTP'
   #define kMyDirectoryType   'SMTP'
   #define kMyDiscriminator   {kMyDirectoryType,0L}

   OSErr             myErr;
   char              *startAddr, *realName;
   RLI               myRLI;
   PackedRLI         myPackedRLI;
   DirDiscriminator  discriminator = kMyDiscriminator;

   CreationID        cid;
   RString           recordName,recordType;
   LocalRecordID     localRID;
   RecordID          RID;

   OCERecipient      theRecipient;
   char              packedRecipient[kMaxRecipSize];
   unsigned long     packedRecipLength;
   RString           xtnValueRString;


   /*
      Not shown here -- parse the address information in the letter from the
      external messaging system. Put the SMTP address into a buffer pointed
      to by startAddr. Put the displayable string that identifies the sender
      or receiver into a buffer pointed to by realName.
   */

   /* make an RLI and check it for validity */
   OCENewRLI(&myRLI, (DirectoryNamePtr)&slotSpec->directoryName,
                        &discriminator, kNULLDNodeNumber, nil);
   if (!OCEValidRLI(&myRLI))
      return kUnexpectedOCECondition;

   /* pack the RLI and check it for validity */
   myErr = OCEPackRLI(&myRLI, &myPackedRLI, kRLIMaxBytes);
```

```
   if (myErr != noErr)
      return myErr;
   if (!OCEValidPackedRLI(&myPackedRLI))
      return kUnexpectedOCECondition;

   /* prepare name and type rstrings and creation ID for local RID */
   OCESetCreationIDtoNull(&cid);             /* set cid to null */
   OCECToRString(realName, smRoman, &recordName, kRStringMaxBytes);
   OCECToRString(kUserRecTypeBody, smRoman, &recordType, kRStringMaxBytes);

   /* the components have been prepared; make the local RID and the RID */
   OCENewLocalRecordID (&recordName, &recordType, &cid, &localRID);
   OCENewRecordID(&myPackedRLI, &localRID, &RID);

   /* build the OCERecipient address structure */
   theRecipient.entitySpecifier  = &RID;
   theRecipient.extensionType    = kMySMTPAddrType;
   OCECToRString(startAddr, smRoman, &xtnValueRString, kRStringMaxChars);
   theRecipient.extensionSize    = xtnValueRString.length+4;
   theRecipient.extensionValue   = (Ptr)&xtnValueRString;

   /* pack the OCERecipient and write it to a disk file */
   packedRecipLength = OCEPackedDSSpecSize(&theRecipient);
   OCEPackDSSpec(&theRecipient, (PackedDSSpec *)&packedRecipient,
                 packedRecipLength);
   myErr = DoWriteAddressToFile(myTempFileSpec, (Ptr)&packedRecipient,
                                packedRecipLength);
}
```

Note

If a personal MSAM receives an incoming letter that contains more than one AOCE recipient, the MSAM translates all of the addresses. However, a personal MSAM cannot forward letters from the user's Macintosh to other AOCE users. A personal MSAM can deliver an incoming letter only to the owner of the local Macintosh computer, even if the letter contains the addresses of other AOCE users. ◆

Logging Personal MSAM Operational Errors

When an operational error occurs, such as a modem not functioning properly or an access number being out of service, the personal MSAM should log the error by calling the `PMSAMLogError` function.

You can log four general classes of information: informational messages, warnings, errors that are not correctable by the user, and errors that are correctable by the user.

These classes are referred to as *error types*; they are represented by four enumerated constants. You use one of these constants in the `errorType` field of the `MailErrorLogEntryInfo` structure when you log an error:

```
enum {
   kMailELECorrectable     = 0,  /* error correctable by user */
   kMailELEError           = 1,  /* error not correctable by user */
   kMailELEWarning         = 2,  /* warning requiring no user intervention */
   kMailELEInformational   = 3   /* informational message */
};
```

For example, you would log an error of type `kMailELEInformational` if you wanted to inform the user that it took 12 connection attempts before a connection with the external messaging system was actually achieved. If you wanted to warn the user that his or her password on the external messaging system was about to expire, you would log an error of type `kMailELEWarning`. You use the `kMailELEError` error type to log an error that cannot be fixed by the user, for example, a missing resource in the personal MSAM. If an error occurs that requires user intervention, you log an error of type `kMailELECorrectable`.

In general, you should log all errors that require user intervention, but you should be selective about logging other types of errors. Logging many warnings and informational messages can fill the error log and cause problems at the user interface.

An error may apply to a specific slot or to the personal MSAM as a whole. When you log an error, you set the `msamSlotID` field of the `MailErrorLogEntryInfo` structure to 0 if the error applies to the personal MSAM as a whole. Otherwise, you set it to the slot ID of the affected slot.

When you log an error of type `kMailELECorrectable`, the IPM Manager considers either the personal MSAM or the affected slot to be suspended. While a personal MSAM is suspended, the IPM Manager does not send it any high-level events or restart it at scheduled times if it quits. While a slot is suspended, the user cannot modify or delete it. Moreover, if you specify the suspended slot in a call to the `PMSAMOpenQueues` function, the function returns the `kMailSlotSuspended` result code. Other than these exceptions, a personal MSAM can continue whatever activity it deems appropriate while it or one of its slots is suspended.

For example, suppose a user configures an SMTP personal MSAM to start up every night at midnight. At midnight, the IPM Manager launches the MSAM, and the MSAM fails to connect to its external messaging system because MacTCP, which is required for this MSAM, is not installed. The MSAM should log an error of type `kMailELECorrectable`. The IPM Manager will not try to launch the SMTP personal MSAM again until the user has installed MacTCP.

Because logging an error of type `kMailELECorrectable` implies that the problem is not transient in nature, the `PMSAMLogError` function does not provide you with a mechanism for canceling these errors or accessing logged entries. Correctable errors, by their definition, require a user's attention, and you should not log them unless absolutely necessary.

AOCE software defines the following error codes:

```
enum {                /* predefined values of MailLogErrorCode */
   kMailMSAMErrorCode   = 0,     /* MSAM-defined error */
   kMailMiscError       = -1,    /* miscellaneous error */
   kMailNoModem         = -2     /* modem required, but missing */
};
```

Because a personal MSAM is a background application, it has no user interface and therefore cannot notify the user of runtime errors. Because each MSAM can potentially encounter errors specific to its implementation, the Finder cannot adequately notify the user of these errors without help from the MSAM. To solve this problem, an MSAM needs to provide two `'STR#'` string list resources. The first `'STR#'` resource contains a list of the MSAM's error messages, each describing a problem that may occur. This resource must have a resource ID of `kMailMSAMErrorStringListID`. The second `'STR#'` resource contains a list of strings specifying the action that the user can take to fix a specific error. It must have a resource ID of `kMailMSAMActionStringListID`.

To cause the Finder to display one of your error messages, you must set the `errorCode` field of the `MailErrorLogEntryInfo` structure to `kMailMSAMErrorCode` and set the `errorResource` field. The `errorResource` field is an index into the list of your error messages in the `'STR#'` resource. The index of the first message in the string list is 1.

When you log an error that requires user intervention (`kMailELECorrectable`), you must specify an action that the user should take to correct the error. You provide the action messages in a `'STR#'` resource (resource ID = `kMailMSAMActionStringListID`). You set the `actionResource` field to an index into the list of your action messages in the `'STR#'` resource. The index of the first message in the string list is 1.

The Finder displays all errors to the user, regardless of the error type. A user reports that an error is corrected by clicking the Resolve button on a problem report in his or her In Tray. (See the *PowerTalk User's Guide* for a description of the PowerTalk user interface.)

The IPM Manager reinstates a suspended personal MSAM or slot when the user reports that the error is corrected or when the computer on which the personal MSAM is running is restarted. If the personal MSAM is not running when the user reports that the problem has been corrected, the IPM Manager launches it. If the personal MSAM is running, it gets a `kMailEPPCContinue` high-level event.

Messaging Service Access Module Reference

This section describes the structures and functions that constitute the messaging service access module API. It also includes descriptions of the high-level events an MSAM might receive.

Data Types and Constants

This section describes the data structures in the MSAM API. The chapters "AOCE Utilities" and "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces* contain descriptions of other structures that you use.

The MSAM Parameter Block

Every function in the MSAM API takes a pointer to an `MSAMParam` parameter block as input. The parameter block has a standard header followed by function-specific fields. Each function description in the section "MSAM Functions" describes the fields of that function's parameter block.

MailParamBlockHeader

The parameter block header for an `MSAMParam` structure has the following definition:

```
# define MailParamBlockHeader
   Ptr            qLink;         /* reserved */\
   long           reservedH1;    /* reserved */\
   long           reservedH2;    /* reserved */\
   ProcPtr        ioCompletion;  /* your completion routine */\
   OSErr          ioResult;      /* result code */\
   long           saveA5;        /* location of app global variables */\
   short          reqCode;       /* reserved */
```

Field descriptions

`qLink`	Reserved.
`reservedH1`	Reserved.
`reservedH2`	Reserved.
`ioCompletion`	Pointer to a completion routine that you can provide. When a function that you called asynchronously completes execution, it calls your completion routine. See page 2-219 for a description of the completion routine. Set this field to `nil` if you do not wish to provide a completion routine. This field is ignored if you call a function synchronously.
`ioResult`	The result of a function. You can poll the `ioResult` field to determine when a function has finished executing. When you execute the function asynchronously, the function sets this field to 1 as soon as the function has been queued for execution. When the function completes execution, it sets this field to the actual result code.
`saveA5`	The contents of your application's A5 register.
`reqCode`	Reserved.

MSAMParam

The `MSAMParam` structure is a union of function-specific substructures, each of which contains standard header fields.

```
union MSAMParam{
   struct {MailParamBlockHeader} header;

   PMSAMGetMSAMRecordPB       pmsamGetMSAMRecord;
   PMSAMOpenQueuesPB          pmsamOpenQueues;
   PMSAMSetStatusPB           pmsamSetStatus;
   PMSAMLogErrorPB            pmsamLogError;
   SMSAMSetupPB               smsamSetup;
   SMSAMStartupPB             smsamStartup;
   SMSAMShutdownPB            smsamShutdown;
   MSAMEnumeratePB            msamEnumerate;
   MSAMDeletePB               msamDelete;
   MSAMOpenPB                 msamOpen;
   MSAMOpenNestedPB           msamOpenNested;
   MSAMClosePB                msamClose;
   MSAMGetMsgHeaderPB         msamGetMsgHeader;
   MSAMGetAttributesPB        msamGetAttributes;
   MSAMGetRecipientsPB        msamGetRecipients;
   MSAMGetContentPB           msamGetContent;
   MSAMGetEnclosurePB         msamGetEnclosure;
   MSAMEnumerateBlocksPB      msamEnumerateBlocks;
   MSAMGetBlockPB             msamGetBlock;
   MSAMMarkRecipientsPB       msamMarkRecipients;
   MSAMnMarkRecipientsPB      msamnMarkRecipients;
   MSAMCreatePB               msamCreate;
   MSAMBeginNestedPB          msamBeginNested;
   MSAMEndNestedPB            msamEndNested;
   MSAMSubmitPB               msamSubmit;
   MSAMPutMsgHeaderPB         msamPutMsgHeader;
   MSAMPutAttributePB         msamPutAttribute;
   MSAMPutRecipientPB         msamPutRecipient;
   MSAMPutContentPB           msamPutContent;
   MSAMPutEnclosurePB         msamPutEnclosure;
   MSAMPutBlockPB             msamPutBlock;
   MSAMCreateReportPB         msamCreateReport;
   MSAMPutRecipientReportPB   msamPutRecipientReport;
   PMSAMCreateMsgSummaryPB    pmsamCreateMsgSummary;
   PMSAMPutMsgSummaryPB       pmsamPutMsgSummary;
   PMSAMGetMsgSummaryPB       pmsamGetMsgSummary;
```

```
   MailWakeupPMSAMPB          wakeupPMSAM;
   MailCreateMailSlotPB       createMailSlot;
   MailModifyMailSlotPB       modifyMailSlot;
};

typedef union MSAMParam MSAMParam;
```

The Mail Buffer

You use the `MailBuffer` structure to pass data between your MSAM and the IPM Manager.

MailBuffer

The mail buffer structure is defined by the `MailBuffer` data type.

```
struct MailBuffer {
   long     bufferSize; /* size of your buffer */
   Ptr      buffer;     /* pointer to your buffer  */
   long     dataSize;   /* amount of data returned in or read out
                           of your buffer */
};

typedef struct MailBuffer MailBuffer;
```

Field descriptions

`bufferSize`	When reading, you set this field to the size of your buffer in bytes. When writing, you set this field to the number of bytes that you want to write.
`buffer`	A pointer to your buffer. You allocate a buffer of whatever size you need.
`dataSize`	When it successfully completes execution, the function sets this field to the actual number of bytes that it read or wrote.

The Mail Reply Structure

A `MailReply` structure is a model. Many functions in the MSAM API format the data they place in a `MailBuffer` structure according to the `MailReply` model format.

MailReply

A structure of type `MailReply` consists of a single field, `tupleCount`, that contains a count. It is followed immediately by `tupleCount` occurrences of a data item or structure. The format of the data item or structure depends on the particular function that returns the data in the `MailReply` structure format. For instance, the `MSAMEnumerate` function returns `MSAMEnumerateOutQReply` or `MSAMEnumerateInQReply` structures.

```
struct MailReply {
   unsigned short tupleCount;
   /* tuple[tupleCount] */
};

typedef struct MailReply MailReply;
```

The Enumeration Structures

The enumeration structures, `MSAMEnumerateOutQReply` and `MSAMEnumerateInQReply`, return information about messages in an outgoing or incoming queue, respectively. The `MSAMEnumerate` function returns a list of one or the other of these structures. Each structure gives enough information about a message for you to know what to do next with the message.

MSAMEnumerateOutQReply

When a personal or server MSAM calls the `MSAMEnumerate` function to enumerate an outgoing queue, the function returns information about the messages in the outgoing queue in a list of `MSAMEnumerateOutQReply` structures, one for each message.

```
struct MSAMEnumerateOutQReply {
   long           seqNum;     /* sequence number of message */
   Boolean        done;       /* resolution of message */
   IPMPriority    priority;   /* priority of message */
   OSType         msgFamily;  /* message family */
   long           approxSize; /* size of message */
   Boolean        tunnelForm; /* reserved */
   Byte           padByte;    /* pad to even byte boundary */
   NetworkSpec    nextHop;    /* reserved */
   OCECreatorType msgType;    /* message creator and type */
};

typedef struct MSAMEnumerateOutQReply MSAMEnumerateOutQReply;
```

Field descriptions

`seqNum`	A sequence number that identifies a specific message in the outgoing queue. It is valid until you delete the message. You pass this value to the `MSAMOpen` function to identify a message you want to open.
`done`	A Boolean value that indicates if you have sent—or completed your attempts to send—the message to each of the recipients for which you are responsible. The IPM Manager sets this field to `true` when you have finished sending or attempting to send the message to all of the recipients for which you are responsible. You tell the IPM Manager which recipients you have processed by calling the `MSAMnMarkRecipients` function.
`priority`	A value that indicates the priority with which the message was sent. Possible values are: `kIPMNormalPriority`, `kIPMLowPriority`, and `kIPMHighPriority`.
`msgFamily`	A value that indicates the message family to which the message belongs. The AOCE-defined message families are `kMailFamily`, `kMailFamilyFile`, and `kIPMFamilyUnspecified`. Developers can define other message families.
`approxSize`	The size of the message itself, not including some overhead bytes associated with the message when it resides in the outgoing queue.
`tunnelForm`	Reserved.
`nextHop`	Reserved.
`msgType`	A structure that specifies the creator and type of the message. The `creator` field indicates the creator of the message. The `type` field identifies the type of message.

MSAMEnumerateInQReply

When a personal MSAM calls the `MSAMEnumerate` function to enumerate an incoming queue, the function returns information about the letters in the queue in a list of `MSAMEnumerateInQReply` structures, one for each letter.

```
struct MSAMEnumerateInQReply {
   long     seqNum;     /* letter sequence number */
   Boolean  msgDeleted; /* should letter be deleted? */
   Boolean  msgUpdated; /* was message summary updated? */
   Boolean  msgCached;  /* is letter in the incoming queue? */
   Byte     padByte;    /* pad to even byte boundary */
};

typedef struct MSAMEnumerateInQReply MSAMEnumerateInQReply;
```

Field descriptions

`seqNum`	A sequence number for a specific letter in the incoming queue. It is valid until you delete the letter.
`msgDeleted`	A Boolean value that indicates whether you should delete the letter. Only the IPM Manager sets and clears this field. If this field is set to `true`, you should delete the letter.
`msgUpdated`	A Boolean value that indicates if the IPM Manager has updated the message summary associated with the letter. Only the IPM Manager sets and clears this field. This field is set to `true` if the IPM Manager has updated the message summary.
`msgCached`	A Boolean value that indicates if the letter is attached to its message summary. Only the IPM Manager sets and clears this field. This field is set to `true` if you wrote the letter into the incoming queue.

The Mail Time Structure

The `MailTime` structure appears in the `sendTimeStamp` attribute in a letter's header and in the `sendTime` field of a letter's message summary.

MailTime

The `MailTime` structure is the standard structure for reporting time in an AOCE system.

```
struct MailTime {
   UTCTime     time;    /* current UTC(GMT) */
   UTCOffset   offset;  /* offset from UTC */
};

typedef struct MailTime MailTime;
```

Field descriptions

`time`	Current time expressed as universal coordinated time (`UTC`) in seconds since 00:00 hours, January 1, 1904. (The `UTCTime` data type is `unsigned long`.)
`offset`	Offset from `UTC` in seconds. The offset is a signed value added to the `time` value. (The `UTCOffset` data type is `long`.)

The Letter Attribute Structures

Letter attributes identify a letter and indicate who wrote it, when it was sent, what its priority for delivery is, who the recipients are, and so forth. Most attributes are stored in the letter header; a few are stored in the message summary.

MailAttributeID

When calling the `MSAMPutAttribute` or `MSAMPutRecipient` function, you use the `MailAttributeID` data type to indicate the letter attribute whose value you are passing to the function. When calling the `MSAMGetRecipients` function, you use it to indicate the recipient type about which you want information.

```
typedef unsigned short MailAttributeID;
```

A variable of type `MailAttributeID` may have any of the following values:

```
enum {
   kMailLetterFlagsBit     = 1,  /* letter flags bit */
   kMailIndicationsBit     = 3,  /* indications bit */
   kMailMsgTypeBit         = 4,  /* letter creator & type bit */
   kMailLetterIDBit        = 5,  /* letter ID bit */
   kMailSendTimeStampBit   = 6,  /* send timestamp bit */
   kMailNestingLevelBit    = 7,  /* nesting level bit */
   kMailMsgFamilyBit       = 8,  /* message family bit */
   kMailReplyIDBit         = 9,  /* reply ID bit */
   kMailConversationIDBit  = 10, /* conversation ID bit */
   kMailSubjectBit         = 11, /* subject bit */
   kMailFromBit            = 12, /* From recipient bit */
   kMailToBit              = 13, /* To recipient bit */
   kMailCcBit              = 14, /* cc recipient bit */
   kMailBccBit             = 15  /* bcc recipient bit */
};
```

MailAttributeBitmap

When calling the `MSAMGetAttributes` function, you use a `MailAttributeBitmap` structure to indicate the letter attributes about which you want information. Each defined bit in the attribute bitmap represents a letter attribute. This structure is also a component part of the `MSAMMsgSummary` structure.

```
struct MailAttributeBitmap {
   unsigned int               /* 32 bits */
      reservedA:16,           /* bits 17 to 32--reserved */
      reservedB:1,            /* bit 16--reserved */
      bcc:1,                  /* bit 15--blind carbon copy recipients */
      cc:1,                   /* bit 14--carbon copy recipients */
      to:1,                   /* bit 13--To recipients */
      from:1,                 /* bit 12--sender of letter */
```

```
        subject:1,              /* bit 11--subject of letter */
        conversationID:1,       /* bit 10--ID of conversation thread */
        replyID:1,              /* bit 09--ID of letter being replied to */
        msgFamily:1,            /* bit 08--message family */
        nestingLevel:1,         /* bit 07--nesting level of letter */
        sendTimeStamp:1,        /* bit 06--time letter was sent */
        letterID:1;             /* bit 05--letter's unique ID number */
        msgType:1,              /* bit 04--letter's creator and type */
        indications:1,          /* bit 03--indications */
        reservedC:1,            /* bit 02--reserved */
        letterFlags:1           /* bit 01--letter flags */
};

typedef struct MailAttributeBitmap MailAttributeBitmap;
```

Field descriptions

`bcc`	Secondary recipients whose addresses do not appear on the letter as received by the To and cc recipients and other bcc recipients.
`cc`	Recipients who are being sent a courtesy copy of the letter.
`to`	Primary recipients of the letter.
`from`	The sender of the letter.
`subject`	The subject of the letter.
`conversationID`	The letter ID number of the original letter that began a sequence of replies or forwards that resulted in the current letter.
`replyID`	The letter ID number of the letter to which the current letter is a reply.
`msgFamily`	A value that indicates the message family to which the message belongs.
`nestingLevel`	The nesting level of the letter. A letter that is newly created (that is, not a reply to or forward of an existing letter) has a nesting level of 0. A reply to or forward of a letter whose nesting level is 0 has a nesting level of 1. A reply to or forward of a letter whose nesting level is 1 has a nesting level of 2, and so on. See the section "Letters" beginning on page 2-17 for information on nested letters.
`sendTimeStamp`	The time the letter was sent.
`letterID`	The letter ID number for the letter. This number is generated by the IPM Manager.
`msgType`	The creator and type of the letter. Each letter has a creator and type.
`indications`	Indications of the properties of the letter, such as whether the letter contains a digital signature, whether the originator requested non-delivery reports, and so on. The `MailIndications` structure is described on page 2-102.
`letterFlags`	Flags that indicate the status of the letter, such as whether it has been opened by the user. The `MailLetterFlags` structure is described on page 2-123. Server MSAMs should ignore this attribute.

The following table summarizes letter attributes. In the column headed "O/M", an *M* indicates *mandatory*—that is, this attribute must always be present. An *O* means *optional*—the attribute may or may not be present in a letter. In the column headed "F/V", an *F* indicates *fixed*—that is, this attribute has a fixed size—while a *V* means *variable*—the attribute size is variable.

Constant	Value	Attribute data type	O/M	F/V
`kMailLetterFlagsBit`	1	`MailLetterFlags`	M	F
`kMailIndicationsBit`	3	`MailIndications`	M	F
`kMailMsgTypeBit`	4	`OCECreatorType`	M	F
`kMailLetterIDBit`	5	`MailLetterID`	M	F
`kMailSendTimeStampBit`	6	`MailTime`	M	F
`kMailNestingLevelBit`	7	`MailNestingLevel`	M	F
`kMailMsgFamilyBit`	8	`OSType`	M	F
`kMailReplyIDBit`	9	`MailLetterID`	O	F
`kMailConversationIDBit`	10	`MailLetterID`	O	F
`kMailSubjectBit`	11	`RString`	O	V
`kMailFromBit`	12	`OCERecipient`	M	V
`kMailToBit`	13	`OCERecipient`	M	V
`kMailCcBit`	14	`OCERecipient`	O	V
`kMailBccBit`	15	`OCERecipient`	O	V

An MSAM should allocate the largest possible buffer for attributes whose size is variable.

Note
All letter attributes except the `letterFlags` attribute are stored in the letter header. Both personal and server MSAMs read or set all letter attributes in the letter header. The `letterFlags` attribute is stored in a letter's message summary. Server MSAMs do not create message summaries and therefore do not set or read a `letterFlags` attribute for letters they handle. The `letterFlags` attribute applies only to letters submitted by a personal MSAM. ◆

MailIndications

The `MailIndications` structure further defines the letter attribute called `indications`. It is a bit field structure that contains information about several characteristics of the letter, such as what priority level the originator set for the letter, whether it has been sent, what type of reports the originator wants, and so on. An MSAM sets many of these bits for an incoming letter and reads the bits for an outgoing letter.

The following constants define bits in the `MailIndications` structure:

```
enum {
   kMailOriginalInReportBit      = 1,
   kMailNonReceiptReportsBit     = 3,
   kMailReceiptReportsBit        = 4,
   kMailForwardedBit             = 5,
   kMailPriorityBit              = 6,
   kMailIsReportWithOriginalBit  = 8,
   kMailIsReportBit              = 9,
   kMailHasContentBit            = 10,
   kMailHasSignatureBit          = 11,
   kMailAuthenticatedBit         = 12,
   kMailSentBit                  = 13
};
```

Note
Constants for the `hasStandardContent`, `hasImageContent`, and `hasNativeContent` bit fields are not defined. ◆

```
struct MailIndications {
unsigned int
   reservedB:16,
   hasStandardContent:1,/* letter has a content block */
   hasImageContent:1,   /* letter has an image block */
   hasNativeContent:1,  /* letter has a content enclosure */
   sent:1,              /* letter sent, not just composed */
   authenticated:1,     /* letter was created and transported with
                           authentication */
   hasSignature:1,      /* letter was signed with a digital signature */
   hasContent:1,        /* this letter or a nested letter has content */
   isReport:1,          /* not a letter, is really a report */
   isReportWithOriginal:1,/* report contains the original letter */
   priority:2,          /* letter has normal, low, or high priority */
   forwarded:1,         /* letter contains a forwarded letter */
   receiptReports:1,    /* originator requests delivery indications */
   nonReceiptReports:1, /* originator requests non-delivery indications */
   originalInReport:2,  /* originator wants original letter enclosed in
                            reports */
};

typedef struct MailIndications MailIndications;
```

Field descriptions

`hasStandardContent`
: If this bit is set, this letter has a block of type `kMailContentType` that contains data in standard interchange format.

`hasImageContent`
: If this bit is set, this letter has a block of type `kMailImageBodyType` that contains data in standard image format.

`hasNativeContent`
: If this bit is set, this letter contains content in the form of a content enclosure.

`sent`
: If this bit is set, this letter was sent, not just composed. This bit is clear for nested letters and those that exist on disk and have not yet been submitted.

`authenticated`
: If this bit is set, this letter was created by an authenticated user and transported over a secure path using the Apple Secure Data Stream Protocol. In release 1, a letter entering an AOCE system via an MSAM is not authenticated. This bit will always be set to 0 on letters read by a personal MSAM. On letters read by a server MSAM, the bit may be set or clear. In either case, it is for the MSAM's information only.

`hasSignature`
: If this bit is set, the sender signed the letter with a digital signature. The signature applies to the letter as a whole. If a portion of the letter is signed, the bit is not set. See the chapter "Digital Signature Manager" in *Inside Macintosh: AOCE Application Interfaces* for information about digital signatures. The AOCE software sets this bit to 0 for letters submitted by an MSAM. If this bit is set for an outgoing letter, the MSAM can ignore it or add a note to the letter indicating that the letter was originally signed with a digital signature.

`hasContent`
: If this bit is set, this letter, or a letter nested within it, contains content. The content can be a content block, an image block, or a content enclosure. Although this bit doesn't indicate the type of content or the nesting level at which the content exists, it provides useful information to AOCE letter applications that display letter content by indicating if a letter has some type of content at some nesting level.

`isReport`
: If this bit is set, this is an IPM report. Because an IPM report is not a report that an MSAM creates or receives, you never set this bit for a report that you create, nor will it be set on a report that you receive. For more information about reports, see the section "Reports" on page 2-23. IPM reports are discussed in the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces*.

`isReportWithOriginal`
If this bit is set, this is an IPM report that contains the original letter to which the report pertains. Because an IPM report is not a report that an MSAM creates or receives, you never set this bit for a report that you create, nor will it be set on a report that you receive. For more information about reports, see the section "Reports" on page 2-23. IPM reports are discussed in the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces*.

`priority` The priority of the letter, as set by the sender. This 2-bit field can be set to any of the following values: `kIPMNormalPriority`, `kIPMLowPriority`, or `kIPMHighPriority`.

```
enum {
   kIPMAnyPriority    = 0,/* not used by MSAM */
   kIPMNormalPriority = 1,
   kIPMLowPriority,
   kIPMHighPriority
};
```

It is up to the recipient to decide how to handle letters of different priorities.

`forwarded` If this bit is set, this letter is a forwarded letter.

`receiptReports` If this bit is set, the originator of this letter has requested a report containing delivery indications.

`nonReceiptReports`
If this bit is set, the originator of this letter has requested a report containing non-delivery indications.

`originalInReport`
This 2-bit field can be set to either of the following values:

```
enum {
   kMailNoOriginal         = 0,
   kMailEncloseOnNonReceipt= 3
};
```

If this field is set to `kMailNoOriginal`, the originator of this letter specified that the original letter not be enclosed in reports. If this field is set to `kMailEnclosedOnNonReceipt`, the originator of this letter specified that the original letter be enclosed in reports containing non-delivery indications. An MSAM ignores this field and never includes a copy of the original letter in a report it creates. The AOCE toolbox is responsible for including originals when appropriate.

The following table indicates who sets the bits in the `MailIndications` structure for an incoming letter. In the column labeled "Responsible for setting," *MSAM* refers to both personal and server MSAMs.

MailIndications bit field	Responsible for setting
`hasStandardContent`	MSAM
`hasImageContent`	MSAM
`hasNativeContent`	MSAM
`sent`	IPM Manager
`authenticated`	IPM Manager
`hasSignature`	IPM Manager
`hasContent`	MSAM
`isReport`	Not applicable
`isReportWithOriginal`	Not applicable
`priority`	MSAM
`forwarded`	MSAM
`receiptReports`	MSAM
`nonReceiptReports`	MSAM
`originalInReport`	MSAM

The Recipient Structures

The structures in this section define the sender or receiver of a message. You use these structures when you get recipient information from a message that you have opened or when you put recipient information into a message that you are creating. The chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces* also describes the `OCERecipient` and `OCEPackedRecipient` structures. The structures are described here from the perspective of an MSAM's use of them.

OCERecipient

The `OCERecipient` structure completely specifies an address. It should contain whatever information is needed to deliver a message to that address.

You use an `OCERecipient` structure to specify a reply address when you call the `MSAMPutMsgHeader` function.

An `OCERecipient` structure is the unpacked form of the `OCEPackedRecipient` structure (described next). The utility routines `OCEPackRecipient` and `OCEUnpackRecipient` allow you to transform the address information from one format to the other. The routines are described in the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces*.

```
struct OCERecipient {
   RecordID*      entitySpecifier;
   OSType         extensionType;
   unsigned short extensionSize;
   Ptr            extensionValue;
};
```

Field descriptions

`entitySpecifier` Pointer to a `RecordID` structure. The record ID contains part of the address. The section "AOCE Addresses" beginning on page 2-23 explains what each field of the `RecordID` structure should contain when it holds either an AOCE address or an external address.

`extensionType` Identifies the type of messaging system with which this recipient is associated. It determines the format and the meaning of the data pointed to by the `extensionValue` field. You must provide an extension type.

`extensionSize` The number of bytes in the `extensionValue` field.

`extensionValue` A pointer to the part of the address that is specific to the messaging system. You should provide the address extension information in an `RString` structure. This allows the information to be displayed properly to the user and allows the user to create new addresses of this type using the type-in addressing feature. (Type-in addressing is a feature of PowerTalk software's human interface.)

Table 2-5 on page 2-30 and Table 2-4 on page 2-29 list the contents of each field in an `OCERecipient` structure for an AOCE address and an external address, respectively.

```
typedef OCERecipient MailRecipient;
```

The `MailRecipient` structure is defined as an `OCERecipient` data type. You use it in exactly the same way as you would an `OCERecipient` structure. You provide a `MailRecipient` structure to specify a recipient of a letter or a report when you call the `MSAMPutRecipient` or `MSAMCreateReport` function, respectively.

OCEPackedRecipient

An `OCEPackedRecipient` structure is the packed form of the `OCERecipient` structure (described in the previous section).

You cannot read the packed address directly. Before you can read it, you must convert it to the unpacked format using the `OCEUnpackRecipient` utility routine. The utility routines `OCESizePackedRecipient`, `OCEGetRecipientType`, and `OCESetRecipientType` allow you to manipulate an `OCEPackedRecipient` structure. They are described in the chapter "Interprogram Messaging Manager" in *Inside Macintosh: AOCE Application Interfaces*.

A structure of type `OCEPackedRecipient` is a minimum-sized structure and should not be allocated on the stack. Instead, use the `NewPtr` or `NewHandle` routine to allocate the structure.

```
struct OCEPackedRecipient {
   unsigned short    dataLength;    /* length of recipient data */
   Byte              data[kOCEPackedRecipientMaxBytes];
};
```

Field descriptions

`dataLength` Length of the packed recipient address that immediately follows this field.

`data` Packed recipient address.

MailOriginalRecipient

The `MailOriginalRecipient` structure consists of a single field, `index`, that contains an index value for a given recipient. The `MailOriginalRecipient` structure is a model of how address information is stored in a buffer. It is always followed immediately by an `OCEPackedRecipient` structure that contains the address information of that recipient. The `MSAMGetRecipients` function returns recipient information in `MailOriginalRecipient` format when you call the function requesting information about recipients of a particular type (From, To, cc, or bcc).

```
struct MailOriginalRecipient {
   short    index;      /* index for recipient */
            /* followed by OCEPackedRecipient structure */
};

typedef struct MailOriginalRecipient MailOriginalRecipient;
```

Field descriptions

`index` An absolute index value associated with the recipient.

MailResolvedRecipient

The `MailResolvedRecipient` structure contains an index value for the recipient, an indication of whether the recipient is a bcc recipient, and a Boolean value that indicates whether you are responsible for delivering the message to this recipient. The `MailResolvedRecipient` structure is a model of how address information is stored in a buffer. The fields of the structure are always followed immediately by an

OCEPackedRecipient structure that contains the address information of the recipient. The MSAMGetRecipients function returns recipient information in MailResolvedRecipient format when you call the function requesting information about resolved recipients.

```
struct MailResolvedRecipient {
   short       index;         /* index for recipient */
   short       recipientFlags;/* recipient information */
   Boolean     responsible;   /* responsible for delivery? */
   Byte        padByte;
               /* followed by OCEPackedRecipient structure */
};

typedef struct MailResolvedRecipient MailResolvedRecipient;
```

Field descriptions

index	An absolute index value associated with the recipient. You need this value when you call the MSAMPutRecipientReport function to identify the recipient to whom the report pertains. The index is also useful if you want to match an original recipient with a resolved recipient.
recipientFlags	A value that tells you if this recipient is a bcc recipient. Use the mask kIPMBCCRecMask to determine if this recipient is a bcc recipient.
responsible	A Boolean value that is set to true if you are responsible for sending the message to this recipient.

The Segment Types

A content block (type kMailContentType) contains the body or main content of a letter in standard interchange format (see the section "Letters" beginning on page 2-17 for more information about interchange format). A content block consists of segments of data in plain text, styled text, picture, sound, or movie format. The MailSegmentType data type identifies one of the five standard data segment types. The MailSegmentMask data type specifies one or more of these segment types. You read and write content blocks with the MSAMGetContent (page 2-150) and MSAMPutContent functions (page 2-186).

MailSegmentType

A variable of the MailSegmentType data type specifies the format of data in a data segment.

```
typedef unsigned short MailSegmentType;
```

A variable of type `MailSegmentType` can contain one of the following values:

```
enum {      /* values of MailSegmentType */
   kMailInvalidSegmentType    = 0,
   kMailTextSegmentType       = 1,
   kMailPictSegmentType       = 2,
   kMailSoundSegmentType      = 3,
   kMailStyledTextSegmentType = 4,
   kMailMovieSegmentType      = 5
};
```

Constant descriptions

`kMailInvalidSegmentType`
: This value is included as a convenience. An MSAM can initialize a variable of type `MailSegmentType` to this known value before calling the `MSAMGetContent` function.

`kMailTextSegmentType`
: The segment contains plain text in one or more character sets. The text data must consist of 1-byte or 2-byte character codes, depending on the character set (Roman, Arabic, Kanji, and so on).

`kMailPictSegmentType`
: The segment contains picture data in PICT format. For more information about PICT format, see *Inside Macintosh: Imaging With QuickDraw*.

`kMailSoundSegmentType`
: The segment contains data in Audio Interchange File Format (AIFF). For more information about AIFF format, see *Inside Macintosh: More Macintosh Toolbox*.

`kMailStyledTextSegmentType`
: The segment contains text and a `StScrpRec` structure containing the style information corresponding to that text. The text data consists of 1-byte or 2-byte character codes, depending on the character set (Roman, Arabic, Kanji, and so on). For more information on the `StScrpRec` structure, the style record, and the style table, see *Inside Macintosh: Text*.

`kMailMovieSegmentType`
: The segment contains QuickTime movie data in QuickTime movie file format ('`MooV`'). For more information about the '`MooV`' file format, see *Inside Macintosh: QuickTime*.

MailSegmentMask

You use the `MailSegmentMask` data type to indicate the kinds of data segments that you want to read when you call the `MSAMGetContent` function.

```
typedef unsigned short MailSegmentMask;
```

The bits in the segment mask are defined as follows:

```
enum{
   kMailTextSegmentBit,
   kMailPictSegmentBit,
   kMailSoundSegmentBit,
   kMailStyledTextSegmentBit,
   kMailMovieSegmentBit
};
```

You can use a combination of the following values to set bits in the segment mask:

```
enum {         /* values of MailSegmentMask */
   kMailTextSegmentMask       = 1L<<kMailTextSegmentBit,
   kMailPictSegmentMask       = 1L<<kMailPictSegmentBit,
   kMailSoundSegmentMask      = 1L<<kMailSoundSegmentBit,
   kMailStyledTextSegmentMask = 1L<<kMailStyledTextSegmentBit,
   kMailMovieSegmentMask      = 1L<<kMailMovieSegmentBit
};
```

The Enclosure Information Structure

You add an enclosure to a letter by calling the `MSAMPutEnclosure` function. The function takes a `MailEnclosureInfo` structure as input. This structure describes the enclosure being added to the letter.

MailEnclosureInfo

You pass a `MailEnclosureInfo` structure to the `MSAMPutEnclosure` function when you enclose a file that resides in memory.

```
struct MailEnclosureInfo {
   StringPtr   enclosureName;
                           /* name of the enclosure */
   CInfoPBPtr  catInfo;    /* HFS catalog info about enclosure*/
   StringPtr   comment;    /* comment for Get Info window */
   Ptr         icon        /* icon for enclosure file */
};

typedef struct MailEnclosureInfo MailEnclosureInfo;
```

Field descriptions

`enclosureName` A pointer to the name of the file that you want to enclose. Format the filename as a Pascal-style string—that is, add a leading length byte. The name must be 1 to 31 bytes long, excluding the length byte, and must not contain colons (:).

`catInfo` A pointer to a fully specified `CInfoPBRec` structure (defined in *Inside Macintosh: Files*), which is returned by the `PBGetCatInfo` function. Set the fields for which you cannot obtain appropriate values to 0, with the exception of the `ioNamePtr` and `ioFlFndrInfo` fields. Ignore the `ioNamePtr` field because you pass the filename in the `enclosureName` field. The first 8 bytes of the `ioFlFndrInfo` field contain values for the file's type and creator. Because the type and creator determine the application associated with the file and the icon that the Finder displays for that file, omitting a value for the `ioFlFndrInfo` field renders the file unusable. Therefore, you should make every attempt to provide meaningful values for the file's creator and type. If you do not know the application associated with the file, set the `creator` field to four question marks ('`????`'). If you do not know the file's type, set the `type` field to ('`????`') as well.

`comment` A pointer to a Pascal-style string containing the file's comment; it is the information that the Get Info command in the Finder displays for the file. The string cannot be longer than 199 characters, excluding the length byte. The Finder truncates a longer string when it places the file on an HFS volume. If the file has no comment, set the `comment` field to `nil`.

`icon` A pointer to the file's icon: the standard black-and-white icon (32 by 32 bits) consisting of 128 bytes of bitmap followed by 128 bytes of mask. Enclosures in a letter are stored in AppleSingle format. AppleSingle format typically provides a single black-and-white icon so that non-Macintosh file systems can easily read an icon without needing to know how to get at the icon resources stored in AppleSingle format. This field preserves compatibility with AppleSingle format. It is not used by AOCE software. You can set this field to `nil`.

The Image Block Information Structure

You use the `TPfPgDir` structure when reading or writing an image block.

TPfPgDir

An image block starts with an image block information structure (the `TPfPgDir` data type defined by the Printing Manager), followed by a series of PICT elements.

```
struct TPfPgDir{
   short iPages;        /* number of pages in image block */
   long  iPgPos[129];   /* array [0..iPfMaxPgs] of offsets */
};
```

Field descriptions

`iPages` The number of pages in the image. The image block contains one PICT for each page.

`iPgPos` An array of offsets from the start of the block to the picture elements that follow the `TPfPgDir` structure.

The `iPgPos` array contains offsets to the picture elements that follow the `TPfPgDir` structure. The offset from the start of the image block to the image of page $n + 1$ is `iPgPos`[n] (because page numbers start at 1 and the array elements start at 0). The array contains `iPgPos`[$n + 1$] elements for a document of n pages. The last element is the offset of the end of the last page from the beginning of the block. You can determine the size of a page by subtracting the offset of the current page from the offset of the next page, that is, the size of page n is `iPgPos`[n] – `iPgPos`[$n – 1$].

The High-Level Event Structures

The `MailEPPCMsg`, `SMCA`, `OCESetupLocation`, `MailLocationFlags`, and `MailLocationInfo` structures are used in conjunction with high-level events.

MailEPPCMsg

When you call the `AcceptHighLevelEvent` function after receiving an AOCE high-level event, the function returns a buffer that contains a `MailEPPCMsg` structure.

```
struct MailEPPCMsg {
   short    version;               /* message version */
   union {
      SMCA *   theSMCA;            /* pointer to SMCA */
      long     sequenceNumber;     /* letter sequence number */
      MailLocationInfo locationInfo;/* location information */
   } u;
};

typedef struct MailEPPCMsg MailEPPCMsg;
```

Field descriptions

`version` The version number of the AOCE high-level event. You should verify that this version number matches the value of the `kMailEPPCMsgVersion` constant in the PowerTalk interface files you used when you built your MSAM.

`u.theSMCA` A pointer to an `SMCA` structure that contains additional information relevant to the event. The IPM Manager uses this field when it sends any of the following events: `kMailEPPCCreateSlot`, `kMailEPPCModifySlot`, `kMailEPPCDeleteSlot`, `kMailEPPCMsgOpened`, `kMailEPPCSendImmediate`, `kMailEPPCAdmin`.

`u.sequenceNumber` The sequence number of the letter to which the event applies. The IPM Manager uses this field when it sends either the `kMailEPPCInQUpdate` or `kMailEPPCDeleteOutQMsg` event.

`u.locationInfo` A `MailLocationInfo` structure. The IPM Manager uses this field when it sends the `kMailEPPCLocationChanged` event.

SMCA

The shared memory communication area, defined by the `SMCA` structure, is used to pass information between the IPM Manager and an MSAM, in addition to the data passed in the `EventRecord` structure.

```
struct SMCA {
   unsigned short smcaLength; /* length of entire SMCA
                                 (including the length field) */
   OSErr          result;     /* result code */
   long           userBytes;  /* event-specific data */
   union{
      CreationID  slotCID;    /* creation ID of record
                                 containing slot information */
      long        msgHint;    /* message reference value */
   } u;
};

typedef struct SMCA SMCA;
```

Field descriptions

`smcaLength` The total length of the `SMCA` structure, including the 2 bytes for the `smcaLength` field itself. The IPM Manager sets this field.

`result` You set this field to acknowledge receipt of the event to the IPM Manager or to indicate that you have handled the event. Set it to the `noErr` result code to acknowledge receipt of the event or to report success. Otherwise, set it to an MSAM-defined error code. See the individual event descriptions for details.

`userBytes`	The interpretation of this field is dependent on the particular event that is being processed. See the individual event descriptions for information on how this field is used for that event.
`u.slotCID`	If the event applies to a particular slot, this field contains the creation ID of the slot's record in the Setup catalog. If the event applies to the MSAM as a whole, this field contains 0. The IPM Manager sets this field. It is irrelevant to server MSAMs.
`u.msgHint`	A reference value associated with a specific letter. The IPM Manager sets this field.

OCESetupLocation

The `OCESetupLocation` data type defines the current system location.

```
typedef char OCESetupLocation;
```

The values 0–8 are valid values for a variable of type `OCESetupLocation`. Values 1–8 refer to an actual location. The value 0 is a special case that indicates the offline or disconnected state. When the current system location is 0, a personal MSAM should not be executing.

The following enumeration defines constants for two of the valid values of type `OCESetupLocation`:

```
enum {
   kOCESetupLocationNone   = 0,  /* disconnect state */
   kOCESetupLocationMax    = 8   /* maximum location value */
};
```

MailLocationFlags

The `MailLocationFlags` data type defines a bit array. Each bit corresponds to a system location. If the bit is set, the slot to which the location flags apply is active at that location. The `MailLocationFlags` data type is used in the `MailLocationInfo` and `MailStandardSlotInfoAttribute` structures.

```
typedef unsigned char MailLocationFlags;
```

A system location is identified by a value ranging from 1 to 8. To test a bit in a variable of type `MailLocationFlags`, the following mask is defined:

```
#define MailLocationMask(locationNumber) (1<<((locationNumber)-1))
```

Note that for the special location value 0, which corresponds to the disconnected or offline state, the mask value is 0. The slot is inactive at all locations when the current system location is 0.

MailLocationInfo

The `MailLocationInfo` structure contains the current system location and a bit array defining the locations at which a given slot is active. The `MailLocationInfo` structure is part of the `MailEPPCMsg` structure. A personal MSAM receives a `MailLocationInfo` structure when it receives a `kMailEPPCLocationChanged` event.

```
struct MailLocationInfo {
   OCESetupLocation     location;   /* the current location */
   MailLocationFlags    active;     /* slot's location flags */
};

typedef struct MailLocationInfo MailLocationInfo;
```

Field descriptions

`location`	A value that identifies the current system location. It may contain any integer value between 0–8.
`active`	A bit array that defines whether or not a given slot is active at each system location.

The Server MSAM Administrative Event Structures

The IPM Manager provides a server MSAM with administrative information by means of the `kMailEPPCAdmin` high-level event (page 2-235).

SMSAMAdminCode

The `SMSAMAdminCode` data type defines a set of codes for server MSAM administrative actions.

```
typedef unsigned short SMSAMAdminCode;
```

A variable of type `SMSAMAdminCode` can have any of the following values:

```
enum {
   kSMSAMNotifyFwdrSetupChange= 1,
   kSMSAMNotifyFwdrNameChange = 2,
   kSMSAMNotifyFwdrPwdChange  = 3,
   kSMSAMGetDynamicFwdrParams = 4
};
```

SMSAMAdminEPPCRequest

The `userBytes` field of the `SMCA` structure associated with a `kMailEPPCAdmin` high-level event provides a pointer to an `SMSAMAdminEPPCRequest` structure. The `SMSAMAdminEPPCRequest` structure contains an administrative code followed by data whose type is determined by the code.

```
struct SMSAMAdminEPPCRequest {
   SMSAMAdminCode       adminCode;          /* admin code */
   union {
      SMSAMSetupChange     setupChange;     /* setup change */
      SMSAMNameChange      nameChange;      /* reserved */
      SMSAMPasswordChange  passwordChange;  /* reserved */
      SMSAMDynamicParams   dynamicParams;   /* reserved */
   } u;
};

typedef struct SMSAMAdminEPPCRequest SMSAMAdminEPPCRequest;
```

Field descriptions

`adminCode` A value that indicates the type of administrative action requested by the `kMailEPPCAdmin` high-level event. The value in this field determines the type of structure contained in the `u` field. In release 1 of PowerTalk system software, this should always be the `kSMSAMNotifyFwdrSetupChange` code.

`u` Contains a structure that varies depending on the value of the `adminCode` field. In release 1 of PowerTalk system software, this should always be an `SMSAMSetupChange` structure.

SMSAMSetupChange

The `SMSAMSetupChange` structure contains connectivity information about a server MSAM.

```
struct SMSAMSetupChange {
   SMSAMSlotChanges  whatChanged;   /* what parameters changed */
   AddrBlock         serverHint;    /* AOCE server address */
};

typedef struct SMSAMSetupChange SMSAMSetupChange;
```

Field descriptions

`whatChanged` A value that indicates the connectivity information that has changed.

`serverHint` The AppleTalk address of the PowerShare catalog server that the MSAM should use to read its Forwarder record containing the changed connectivity information. Because an AOCE system is a distributed system, the changed data may not have propagated to other servers yet.

SMSAMSlotChanges

The `SMSAMSlotChanges` data type defines a bit array that indicates the kind of connectivity information that has changed.

```
typedef unsigned long SMSAMSlotChanges;
```

The bits in the `SMSAMSlotChanges` data type are defined as follows:

```
enum {
   kSMSAMFwdrHomeInternetChangedBit,
   kSMSAMFwdrConnectedToChangedBit,
   kSMSAMFwdrForeignRLIsChangedBit,
   kSMSAMFwdrMnMServerChangedBit
};
```

You can use the following values to test the bits in a variable of type `SMSAMSlotChanges`:

```
enum {          /* values of SMSAMSlotChanges */
   kSMSAMFwdrEverythingChangedMask  = -1,
   kSMSAMFwdrHomeInternetChangedMask= 1L<<kSMSAMFwdrHomeInternetChangedBit,
   kSMSAMFwdrConnectedToChangedMask = 1L<<kSMSAMFwdrConnectedToChangedBit,
   kSMSAMFwdrForeignRLIsChangedMask = 1L<<kSMSAMFwdrForeignRLIsChangedBit,
   kSMSAMFwdrMnMServerChangedMask   = 1L<<kSMSAMFwdrMnMServerChangedBit
};
```

Constant descriptions

`kSMSAMFwdrEverythingChangedMask`
In release 1 of the AOCE software, this constant has the same definition as that of the `kSMSAMFwdrForeignRLIsChangedMask` constant.

`kSMSAMFwdrHomeInternetChangedMask`
Reserved.

`kSMSAMFwdrConnectedToChangedMask`
Reserved.

`kSMSAMFwdrForeignRLIsChangedMask`
The record location information that points to a catalog associated with the MSAM's external messaging system has changed. The information changes when the PowerShare system administrator adds or deletes a catalog for a messaging system served by the MSAM.

`kSMSAMFwdrMnMServerChangedMask`
Reserved.

The Personal MSAM Setup Structures

The `MailTimer` and `MailTimerKind` data types and the `MailTimers` and `MailStandardSlotInfoAttribute` structures contain the user's send and receive requirements for a given slot and location information for that slot.

MailTimer

A variable of type `MailTimer` specifies a number of seconds. The value is interpreted as a frequency interval or a specific time, depending on which union field is used.

```
union MailTimer {
   long  frequency;     /* how often to connect */
   long  connectTime;   /* time since midnight */
};

typedef union MailTimer MailTimer;
```

Field descriptions

`frequency` A value that tells a personal MSAM how often it should connect to its messaging system to send or retrieve mail. The frequency interval is specified in seconds.

`connectTime` A value that tells a personal MSAM at what time it should connect to its messaging system to send or retrieve mail. The time is specified as the number of seconds since midnight. The midnight used is that of the internal time on the Macintosh as set by the user.

MailTimerKind

A variable of type `MailTimerKind` specifies the type of timer that a user wants to use with a given mail slot.

```
typedef Byte MailTimerKind;
```

A variable of type `MailTimerKind` can have any of the following values:

```
enum {
   kMailTimerOff        = 0,  /* no timer specified */
   kMailTimerTime       = 1,  /* timer relative to midnight */
   kMailTimerFrequency  = 2   /* frequency timer*/
};
```

Constant descriptions

`kMailTimerOff` Specifies that the user has not requested a timer.

`kMailTimerTime` Specifies that a personal MSAM should send or retrieve messages at a particular time.

`kMailTimerFrequency` Specifies that a personal MSAM should send or retrieve messages at regular intervals.

MailTimers

The `MailTimers` structure indicates how frequently a personal MSAM connects to its external messaging system. A personal MSAM's setup template sets the fields of the `MailTimers` structure in response to user actions. The user can express the frequency as a particular clock time at which the personal MSAM automatically connects every day (for example, connect at 3:00 A.M. to send and receive letters) or as a periodic occurrence (for example, connect every two hours). The IPM Manager uses the information in this structure to determine when it should send a `kMailEPPCSchedule` event to the personal MSAM.

```
struct MailTimers {
   MailTimerKind  sendTimeKind;     /* timer kind for sending */
   MailTimerKind  receiveTimeKind;  /* timer kind for receiving */
   MailTimer      send;             /* connect time or frequency
                                       for sending letters */
   MailTimer      receive;          /* connect time or frequency
                                       for receiving letters */
};

typedef struct MailTimers MailTimers;
```

Field descriptions

`sendTimeKind` A constant that indicates what type of timer the user wants the personal MSAM to use for sending messages for a particular slot. The setup template sets this field to one of the following values: `kMailTimerTime`, `kMailTimerFrequency`, or `kMailTimerOff`.

`receiveTimeKind`
: A constant that indicates what type of timer the user wants the personal MSAM to use for retrieving messages for a particular slot. The setup template sets this field to one of the following values: `kMailTimerTime`, `kMailTimerFrequency`, or `kMailTimerOff`.

`send`
: A value that specifies either the time interval that elapses before the personal MSAM sends messages to its external messaging system or a specific time at which the MSAM sends these messages. The MSAM interprets this field according to the value in the `sendTimeKind` field. If that value is `kMailTimerOff`, the MSAM ignores this field.

`receive`
: A value that specifies either the time interval that elapses before the personal MSAM retrieves messages from its external messaging system or a specific time at which the MSAM retrieves these messages. The MSAM interprets this field according to the value in the `receiveTimeKind` field. If that value is `kMailTimerOff`, the MSAM ignores this field.

MailStandardSlotInfoAttribute

The personal MSAM's setup template obtains location and timing information from the user to set the `active` and `sendReceiveTimer` fields of this structure appropriately. Then it adds the structure to the slot's Combined or Mail Service record in the Setup catalog, where the information is available to the IPM Manager.

```
struct MailStandardSlotInfoAttribute {
      short             version;       /* version of this slot structure */
      MailLocationFlags active;        /* active at location i if
                                          MailLocationMask(i) is set */
      Byte              padByte;
      MailTimers        sendReceiveTimer;
};

typedef struct MailStandardSlotInfoAttribute MailStandardSlotInfoAttribute;
```

Field descriptions

`version`
: The version of the `MailStandardSlotInfoAttribute` structure. You should set this field to 1. There is no constant defined for it.

`active`
: A bit array that defines whether or not the slot is active at a given location. If the bit is set, the slot is active at the corresponding location. A slot is active if a personal MSAM is able to send and receive messages for the slot.

`sendReceiveTimer`
: The frequency at which the IPM Manager should schedule the personal MSAM to send and receive messages for the user account represented by this slot. (The IPM Manager does this by sending the MSAM a `kMailEPPCSchedule` event.)

The Personal MSAM Letter Flag Structures

The letter flags provide information about a letter in an incoming queue. Only personal MSAMs use the structures in this section.

MailLetterSystemFlags

The IPM Manager sets the letter system flags.

```
typedef unsigned short  MailLetterSystemFlags;
```

The bit in the system flags bytes that you can test is defined as follows:

```
enum {
   kMailIsLocalBit = 2
   };
```

You can use the following value to test the bit flag in the `MailLetterSystemFlags` data type.

```
enum {
   kMailIsLocalMask        = 1L<<kMailIsLocalBit
};
```

Constant descriptions

`kMailIsLocalMask`

The letter exists in an incoming queue on the local computer. If the `kMailIsLocalBit` bit is not set, the letter is stored on an external messaging system, and only its message summary is currently available locally.

MailLetterUserFlags

The IPM Manager and a personal MSAM can set letter user flags in response to a user action.

```
typedef unsigned short  MailLetterUserFlags;
```

The bits in the user flags bytes are defined as follows:

```
enum{
   kMailReadBit,
   kMailDontArchiveBit,
   kMailInTrashBit
   };
```

You can use the following values to test the flags in the `MailLetterUserFlags` data type.

```
enum {
   kMailReadMask           = 1L<<kMailReadBit,
   kMailDontArchiveMask    = 1L<<kMailDontArchiveBit,
   kMailInTrashMask        = 1L<<kMailInTrashBit
};
```

Constant descriptions

`kMailReadMask` The user has opened this letter. A personal MSAM sets the letter user flags to 0 when it creates the letter's message summary. The IPM Manager sets the `kMailReadBit` bit to 1 when the user opens the letter. A personal MSAM can also modify this bit by calling the `PMSAMPutMsgSummary` function.

`kMailDontArchiveMask` Reserved.

`kMailInTrashMask` Reserved.

MailLetterFlags

The `MailLetterFlags` structure contains both system and user letter flags to indicate the status of a letter.

```
struct MailLetterFlags {
   MailLetterSystemFlags   sysFlags;   /* system flags */
   MailLetterUserFlags     userFlags;  /* user flags */
};

typedef struct MailLetterFlags MailLetterFlags;
```

Field descriptions

`sysFlags` A set of bit flags managed by the IPM Manager. You can test the `kMailIsLocalBit` bit to determine if a given letter is actually stored on the local computer.

`userFlags` A set of bit flags that indicate state changes that are controlled by the user. The only bit flag that is relevant to an MSAM is the `kMailReadBit` bit, which indicates whether the user has opened the letter. You can test this bit with the `kMailReadMask` constant.

MailMaskedLetterFlags

Use the `MailMaskedLetterFlags` structure to set the letter flags attribute in a letter. This structure is used by the `MSAMPutMsgSummary` function.

```
struct MailMaskedLetterFlags {
   MailLetterFlags   flagMask;   /* flags that are to be set */
   MailLetterFlags   flagValues; /* their values */
};

typedef struct MailMaskedLetterFlags MailMaskedLetterFlags;
```

Field descriptions

`flagMask`	The flags that are to be set.
`flagValues`	The values of the flags that you want to set.

The Personal MSAM Message Summary Structures

A personal MSAM creates a message summary to store summary information about a letter. The Finder uses message summary information to display incoming letters to the user. The `MSAMMsgSummary` structure defines a message summary. A message summary consists of a few individual fields and two groups of letter attributes. The two groups of letter attributes are defined by the `MailMasterData` and `MailCoreData` structures, described in this section.

MailMasterData

The attributes specified in the `MailMasterData` structure are not critical to the Finder when it displays information about the letter to which the message summary belongs.

```
struct MailMasterData {
   MailAttributeBitmap  attrMask;      /* indicates attributes present in
                                          letter */
   MailLetterID         messageID;     /* ID of this letter *
   MailLetterID         replyID;       /* ID of letter this is a reply to */
   MailLetterID         conversationID;/* ID of letter that started this
                                          conversation */
};

typedef struct MailMasterData MailMasterData;
```

Field descriptions

`attrMask`	A bit array that indicates letter attributes. You must set the bits that correspond to the attributes that are present in the letter. See the description of the `MailAttributeBitmap` structure on page 2-100 for a description of the bits in the attribute bitmap.
`messageID`	The letter ID of this letter. The letter ID is a value that uniquely identifies the letter. The letter ID is provided by the IPM Manager.
`replyID`	The letter ID of the letter to which this letter is a reply. You provide this value if it exists in the letter.
`conversationID`	The letter ID of the original letter that began a sequence of replies or forwards that resulted in this letter. You provide this value if it exists in the letter.

MailCoreData

The Finder uses the attributes specified in the `MailCoreData` structure when it displays information about the letter to which the message summary belongs. You provide values for the fields of the structure, except where otherwise noted in the field descriptions.

```
/* defines for the addressedToMe field */
#define kAddressedAs_TO 0x1
#define kAddressedAs_CC 0x2
#define kAddressedAs_BCC 0x4

struct MailCoreData {
   MailLetterFlags      letterFlags;   /* letter status flags */
   unsigned long        messageSize    /* size of letter */
   MailIndications      letterIndications;
                                       /* indications for this letter */
   OCECreatorType       messageType;   /* message creator and type of this
                                          letter */
   MailTime             sendTime;      /* time this letter was sent */
   OSType               messageFamily; /* message family */
   unsigned char        reserved;
   unsigned char        addressedToMe; /* user is To, cc, or bcc recipient */
   char                 agentInfo[6];  /* reserved (set to 0) */
   /* these are variable length and even padded */
   RString32            sender;        /* sender of this letter */
   RString32            subject;       /* subject of this letter */
};

typedef struct MailCoreData MailCoreData;
```

Field descriptions

`letterFlags` A set of bit flags that indicate the status of the letter, such as whether it has been opened by the user. Set this field to 0. See the description of the `MailLetterFlags` structure on page 2-123 for more information on these bit flags. You can modify the user portion of the letter flags when you call the `PMSAMPutMsgSummary` function.

`messageSize` The size of the letter in bytes. You provide this value.

`letterIndications` Indications of additional properties of the letter, such as whether the letter contains a digital signature, whether or not the originator requested non-delivery indications, and so on. See the description of the `MailIndications` structure on page 2-102. You provide this value.

`messageType` The creator and type of the letter. Every letter has a creator and type. You must provide this value.

`sendTime` The time the letter was sent. You provide this value.

`messageFamily` A value that indicates the message family to which the message belongs. Set this field to `kMailFamily`.

`reserved` Reserved.

`addressedToMe` Indicates how the letter was sent to the addressee: as a To address, a cc address, or a bcc address; possible values are `kAddressedAs_TO`, `kAddressedAs_CC`, and `kAddressedAs_BCC`. You must set this field appropriately. You can set more than one bit.

`agentInfo` Reserved. Set this field to 0.

`sender` The sender of the letter. You must provide a value for this field. If your sender information consists of an odd number of bytes, add a pad byte so that it ends on an even byte boundary. The IPM Manager treats this field and the `subject` field that follows as a single common buffer that contains variable-length sender and subject information. See the section "Creating a Letter's Message Summary" beginning on page 2-64 for information on how to correctly assign a value to this field.

`subject` The subject of the letter. You must provide this value. If your subject information consists of an odd number of bytes, add a pad byte so that it ends on an even byte boundary. The IPM Manager treats this field and the `sender` field before it as a single common buffer that contains variable-length sender and subject information. You add the subject on the first even-byte boundary following the sender information, which is not necessarily the same as the beginning of this field. See the section "Creating a Letter's Message Summary" beginning on page 2-64 for information on how to correctly assign a value to this field.

MSAMMsgSummary

An `MSAMMsgSummary` structure provides summary information about an incoming letter. You must create one of these structures for each incoming letter. (In addition to the fields defined in the message summary structure, the IPM Manager stores up to `kMailMaxPMSAMMsgSummaryData` bytes of MSAM-specific private data with a message summary.)

```
struct MSAMMsgSummary {
   short                version;       /* version of the MSAMMsgSummary
                                          structure */
   Boolean              msgDeleted;    /* should letter be deleted? */
   Boolean              msgUpdated;    /* was message summary updated? */
   Boolean              msgCached;     /* is letter in the incoming queue? */
   Byte                 padByte;
   MailMasterData       masterData;    /* attributes not essential to
                                          display */
   MailCoreData         coreData;      /* attributes critical to display *
};

typedef struct MSAMMsgSummary MSAMMsgSummary;
```

Field descriptions

`version` The version of the message summary structure. You must set this field to the constant `kMailMsgSummaryVersion`.

`msgDeleted` A Boolean value indicating whether you should delete this letter. You do not provide a value for this field. The IPM Manager initially sets this field to `false`. It sets this field to `true` when the user deletes a letter. If this field is `true`, you should delete the letter on your external messaging system and delete the letter's message summary.

`msgUpdated` A Boolean value indicating whether the IPM Manager updated information in the message summary. You do not provide an initial value for this field. The IPM Manager initially sets this field to `false`. It sets this field to `true` when it updates any of the following fields in the message summary: `msgDeleted`, `msgStoreFlags`, `finderInfo`. You read this field to determine if the message summary has changed. If it has, you should reexamine the message summary and take appropriate action, if any, based on the changed information. After taking the action, you should reset this field to `false`.

`msgCached` A Boolean value indicating whether the letter associated with the message summary exists in an incoming queue. You do not provide a value for this field. The IPM Manager initially sets this field to `false`. It sets this field to `true` when you write the letter corresponding to this message summary into the incoming queue.

`masterData`	A `MailMasterData` structure that contains letter attributes not essential to the ability of the Finder to display the letter. See the structure description on page 2-124 for an explanation of the information that you must provide.
`coreData`	A `MailCoreData` structure that contains the attributes crucial to the Finder's ability to display the letter. See the structure description on page 2-125 for an explanation of the information that you must provide.

The Personal MSAM Error Log Entry Structure

The error log is where a personal MSAM can report errors that require a user's intervention to correct. The personal MSAM reports errors using the `PMSAMLogError` function. The function takes a pointer to a `MailErrorLogEntryInfo` structure as input.

MailErrorLogEntryInfo

You provide a `MailErrorLogEntryInfo` structure to the `PMSAMLogError` function when you want to report an operational error to the IPM Manager and ultimately to the user.

```
typedef unsigned short MailLogErrorType;

/* values of MailLogErrorType */
enum {
   kMailELECorrectable  = 0,  /* error correctable by user */
   kMailELEError        = 1,  /* error not correctable by user */
   kMailELEWarning      = 2,  /* warning requiring no user intervention */
   kMailELEInformational= 3   /* informational message */
};

typedef short MailLogErrorCode;

/* predefined values of MailLogErrorCode */
enum {
   kMailMSAMErrorCode   = 0,     /* MSAM-defined error */
   kMailMiscError       = -1,    /* miscellaneous error */
   kMailNoModem         = -2     /* modem required, but missing */
};

struct MailErrorLogEntryInfo {
   short             version;             /* log entry version */
   UTCTime           timeOccurred;        /* time of error */
   Str31             reportingPMSAM;      /* MSAM reporting the error */
   Str31             reportingMSAMSlot;   /* slot having the error */
```

```
   MailLogErrorType  errorType;          /* level of error */
   MailLogErrorCode  errorCode;          /* error code */
   short             errorResource;      /* error string resource index */
   short             actionResource;     /* action string resource index */
   unsigned long     filler;             /* reserved */
   unsigned short    filler2;            /* reserved */
};

typedef struct MailErrorLogEntryInfo MailErrorLogEntryInfo;
```

Field descriptions

`version` The version of the error log entry. Set this field to `kMailErrorLogEntryVersion`.

`timeOccurred` The time that the error occurred. This is filled in by the IPM Manager.

`reportingPMSAM` A string identifying the personal MSAM that is logging the error. This is filled in by the IPM Manager.

`reportingMSAMSlot` A string identifying the slot that is experiencing the error, if the error is associated with a specific slot. This is filled in by the IPM Manager.

`errorType` A value that indicates the type of error that you are logging. Set this field to one of the following constants: `kMailELECorrectable`, `kMailELEError`, `kMailELEWarning`, `kMailELEInformational`.

`errorCode` A value that indicates the error you are logging. There are three predefined errors; you can define others. If you want to log an error that you define, set this field to `kMailMSAMErrorCode` and set the `errorResource` field to the index into your string list (`'STR#'`) resource for the string that describes the error. The constants for the predefined errors are `kMailMSAMErrorCode`, `kMailMiscError`, and `kMailNoModem`.

`errorResource` An index into your list of error messages. An error message describes the problem that has occurred. The resource ID of the `'STR#'` resource containing the list of error messages must be `kMailMSAMErrorStringListID`. If you are logging an AOCE-defined error, the IPM Manager ignores this field.

`actionResource` The index into your list of action messages. An action message is always associated with an error of type `kMailELECorrectable`. The action message recommends the action that the user should take to correct the error. The resource ID of the `'STR#'` resource containing the list of action messages must be `kMailMSAMActionStringListID`. If you are logging an AOCE-defined error, the IPM Manager ignores this field.

See the section "Logging Personal MSAM Operational Errors" on page 2-91 for more information about operational errors.