Interprogram Messaging Manager

This chapter describes the AOCE Interprogram Messaging (IPM) Manager. The IPM Manager provides a low-level interface to the AOCE store-and-forward messaging service.

You can use the IPM Manager to send a message from one AOCE-aware application to another. There are no restrictions on the contents of AOCE interprogram messages. However, if you want to send or read messages intended to be read by people, you should use the Standard Mail Package instead of the IPM Manager. Such messages are referred to as *letters*. The Standard Mail Package provides a high-level interface to the AOCE store-and-forward messaging service specifically to support letters. It is described in the chapter "Standard Mail Package" in this book.

This chapter assumes that you are familiar with AOCE catalog concepts, including catalog records, attribute types, and attribute values, as described in the chapter "Catalog Manager" in this book.

This chapter provides an introduction to AOCE interprogram messages and the IPM Manager and then discusses how you can use the IPM Manager to

- create and send a message to one or more recipients
- manage the queues in which the IPM Manager places messages
- list and read the messages that you receive

About the IPM Manager

The Apple Open Collaboration Environment provides a store-and-forward messaging service that can deliver a message from one application to another regardless of whether the applications are simultaneously connected to a network, or, in fact, regardless of whether they are connected to a network at all. In addition to general application-to-application messages, the Apple Open Collaboration Environment defines a special category of messages, called *letters*, that are intended to be read by people. The sending and receiving of letters by AOCE-aware applications is referred to as the *AOCE mail service*. The IPM Manager provides a low-level interface to AOCE messaging services. The Standard Mail Package is a client of the IPM Manager that provides a high-level interface to AOCE mail services.

The IPM Manager application interface is the same no matter what transport medium is being used to carry the message. Apple Computer, Inc., provides interfaces between the IPM Manager and an AppleTalk network with and without a mail and messaging server. Apple also provides the Direct Dialup mail and messaging service access module (MSAM), which allows the IPM Manager to use a modem to send messages over telephone lines. Other developers can provide MSAMs that allow the IPM Manager to use other transport media and messaging services, such as Ethernet networks or fax modems.

The IPM Manager maintains output and input queues on the local hard disk to store messages waiting to be forwarded or to be read. The IPM Manager can use the output queue, for example, to store a message until the telephone-connection MSAM can establish a modem-to-modem connection. Any number of applications can use the same queue. You can ask for a list of messages filtered by creator, so you need not sort through all of the messages intended for other applications. However, if you have a need to do so, you can also create any number of input queues for the use of your application.

When you send a message, you must specify the addresses of one or more recipients. If a recipient or group of recipients has an associated record in an AOCE catalog, you can specify the record ID and the attribute containing the address, and the IPM Manager looks up the address in the catalog. Alternatively, you can specify the type of connection and provide specific information about the address of the recipient, such as the telephone number and modem information or the AppleTalk network address.

You can use the IPM Manager to

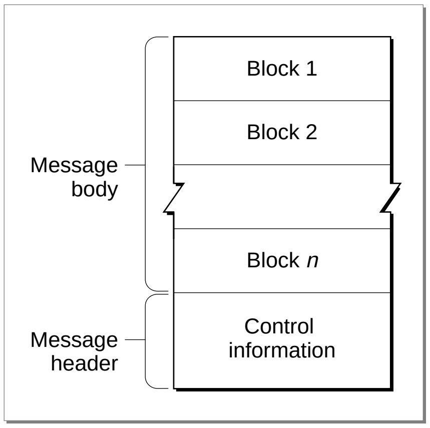
- create a new message
- add blocks to a message
- write data to a message block
- nest a message within a message
- address a message
- send a message or save it to a disk file
- create input queues
- open input queues
- obtain a list of received messages
- filter received-message lists by such attributes as priority, message type, or script code
- read message-header information
- read message blocks
- delete messages from an input queue
- close input queues

About AOCE Interprogram Messages

The AOCE store-and-forward messaging service implemented by the IPM Manager uses messages that consist of a header plus any number of message blocks. The header contains addressing information, a table of contents of the message blocks, other information of interest to the receiving application (such as the message type and priority), and information used solely by the IPM Manager. Each message block can be of any length less than 2^{32} bytes and can contain any type of data. Apple Computer has defined a few message types and message block types, such as the standard-letter-content block type used by the Standard Mail Package. You can define any message block types you wish.

Figure 7-1 illustrates the basic structure of a message. Note that the message header is actually located at the end of the message, after all the message blocks.

Figure 7-1 Structure of an AOCE message



When a block contains a message, the message inside the block is called a ***nested message.*** A message can contain any number of nested messages, and any nested message can contain other nested messages. The structure of a nested message is exactly the same as the structure of a message. Figure 7-2 illustrates a message containing a nested message.

Note
If you are using the IPM Manager to send letters to the Standard Mail Package, you should avoid sending any nested letters that contain standard content. If the Standard Mail Package receives a letter that contains a nested letter, it ignores any content (standard interchange format or image format) within the nested letter. It displays the header and nesting information of the nested letter as a forwarded mailer. ◆

Figure 7-2 An AOCE message containing a nested message

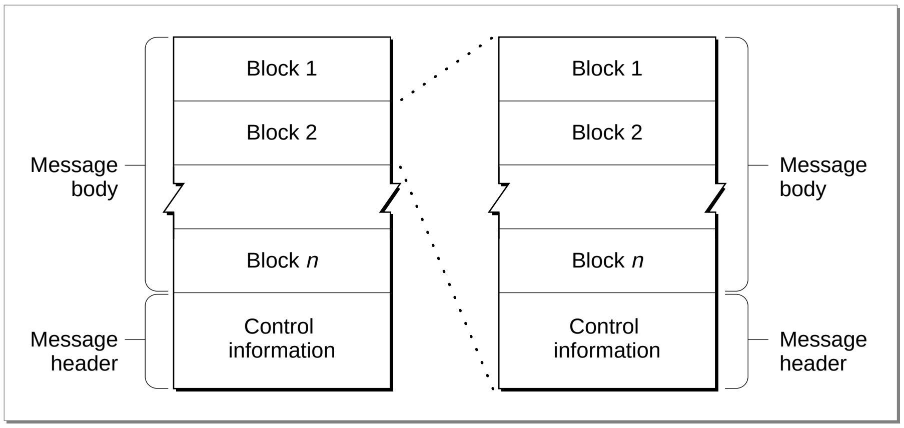
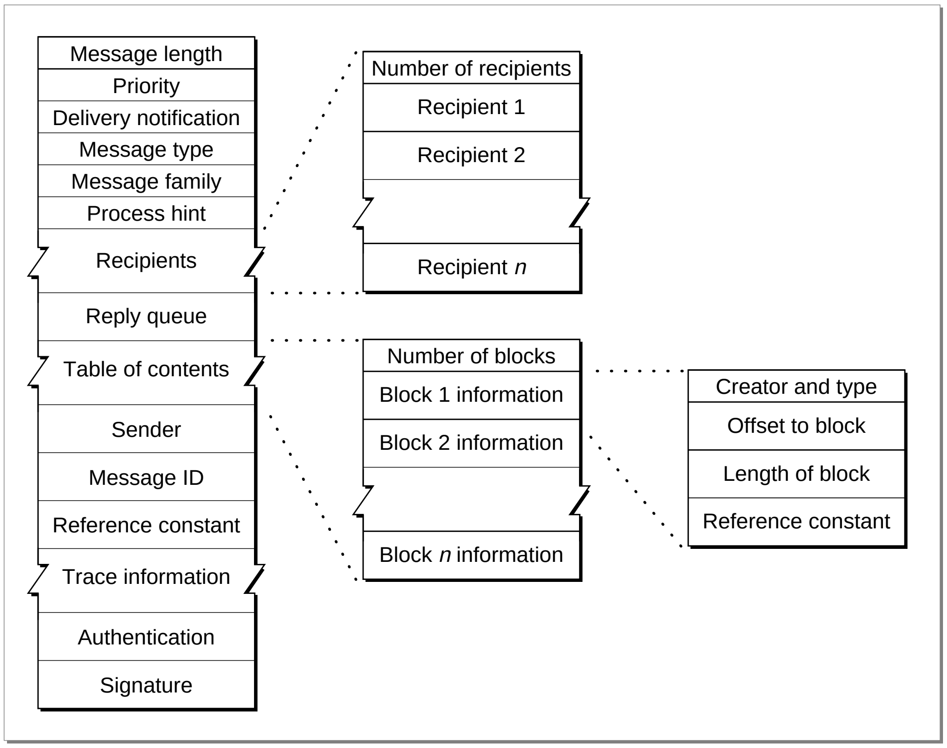


Figure 7-3 illustrates the contents of a message header. Note that Figure 7-3 does not show the size or true sequence of fields in the message header. You must use IPM Manager routines to read and write message-header information.

Figure 7-3 Contents of an AOCE message header



Although all of the public message-header fields are described in detail in the reference section of this chapter, several fields of general interest are briefly described here.

The sender of a message assigns a priority (low, normal, or high) to it. The IPM Manager does not read the contents of the priority field; it is up to the receiving application to determine how to handle messages of different priorities.

When you send a message, you can request delivery and nondelivery reports. The delivery notification field in the message header tells the IPM Manager what kinds of reports you want to receive. Reports are AOCE messages and can include the original message as a nested message if you request that option. Report messages are described in "Report Messages" on page 7-9.

The message type consists of a creator field and a type field. The sending application assigns the message type, and the receiving application uses it to help determine how to interpret the contents of the message. Apple Computer has defined some standard message types for report messages and letters. You can define other message types for whatever purpose you wish.

The message family is a class of messages. Apple Computer has defined some standard message family types for mail and reserves all message family types consisting entirely of lowercase characters. You can define your own message family types, but Apple Computer does not register or otherwise control developer-defined message family types.

The process hint is a character string that you can use for any purpose, such as discriminating among subtypes of messages of the same type or internal routing of messages.

When you send a message, you must specify location information for each recipient. You can specify the record ID of a user record if the recipient's address is stored in an AOCE catalog, or you can specify the actual delivery address of the recipient.

The reply queue is the address to which the IPM Manager should return delivery and nondelivery reports and to which reply messages should be sent.

The table of contents specifies the type and location of each block. The block type includes a creator field and a type field. Apple Computer has defined some standard block types for such things as nested messages and standard-letter-content blocks. You can define other block types for your own use.

In the case of an authenticated message, the sender field is filled in by the IPM Manager and identifies the authenticated originator of the message. In the case of an unauthenticated message, such as a message sent over a serverless network or over a dialup connection, the originator of the message fills in the sender field. In this case, the field should give some indication of who originated the message, but the IPM Manager can not ensure its accuracy or usefulness.

The reference constant is a numeric reference value that the creator of the message provides for the message. You might use this field, for example, to indicate that the message includes blocks of a certain type so that the receiving application can allocate the memory resources it will need to read the message. The table of contents (TOC information) for each block also contains a reference constant that you can use for any purpose you wish.

The IPM Manager sets the authentication information field to indicate whether the message was sent over a secure, authenticated connection. In the case of a message that passes through more than one store-and-forward server, the IPM Manager sets this field to `true` only if the identities of the original sender and of every server in the routing chain were authenticated. The authentication field does not reflect the authentication status of the communication link that the addressee uses to read the message from the last server's message queue. The chapter "Authentication Manager" in this book describes the authentication process in detail.

If the sending application adds a digital signature to a message, the IPM Manager adds a signature block to the message and sets the signature field of the message header to `true`.

Message Queues

The IPM Manager delivers a message to a ***message queue,*** which is maintained by the IPM Manager on the recipient's disk or by a server on the disk of the server computer. Any application can create message queues. Before you can list the messages in a message queue or read a message in a queue, you must open the queue.

Each queue can be opened any number of times, by any number of applications. Each time an application opens a queue the IPM Manager assigns a queue reference number. Each time you list the messages in the queue, open a message, read information from a message, close a message, or delete a message, you must specify a queue reference number.

When you list the messages in a queue, you can specify a filter that limits the messages included in the list. For example, you can filter a queue list for messages with a specific creator to limit it to messages sent by your own application. You can also filter queue lists by message priority or process hint (an application-defined value). When you open a queue (and so obtain a queue reference number), you can specify a default queue filter to be associated with that queue reference number. You can change the default queue filter at any time.

If you open a queue three times to get three queue reference numbers, it appears as though you have three queues, especially if you specify a different queue filter each time you open the queue. Note, however, that these three "queues" are all actually views of the same physical queue and so may list some or all of the same messages. To distinguish between the queue on disk and the apparent queues you get when you open the queue, this book refers to the ***physical queue*** on disk and to ***virtual queues*** associated with that physical queue. Each queue reference number identifies one virtual queue. A physical queue can have any number of associated virtual queues. When you close a virtual queue, the IPM Manager automatically closes all the messages that were opened through that virtual queue.

You can use a virtual queue to open and close messages regardless of whether the same messages are already open through another virtual queue. However, when you *delete* a message, it is deleted from the physical queue and so from all the virtual queues associated with that physical queue. (The IPM Manager prevents you from deleting a message as long as it is open through any virtual queue.)

The primary reason the IPM Manager provides virtual queues is to allow more than one application to use the same physical queue simultaneously. However, you can also use virtual queues to help organize your bookkeeping. You can use multiple virtual queues as a convenient way to group messages, especially if your message groups are based on message type or creator, script code, priority, or process hint.

For example, an application for stockbrokers might receive two types of IPM messages: notices about stock prices and orders sent by clients. Such an application might maintain two virtual queues to make it easier to list, open, and close the two message types independently.

In much the same way that virtual queues link together messages that you might want to list, open, or close together, each virtual queue is associated with a ***queue context.*** You must open at least one queue context before you can open a queue, and each time you open a queue you must specify to which context the virtual queue is to belong. When you close a queue context, the IPM Manager automatically closes all of the queues associated with that context. If you are using several virtual queues to organize messages, you might want to use more than one queue context to add another hierarchical level to the organization.

To extend the previous illustration, for example, suppose the stockbrokers' application has separate virtual queues for low-, normal-, and high-priority buy-or-sell orders, and links these three queues together by assigning them all to the same context. Then the application could close all the high-priority orders by closing one virtual queue, or it could close all of the orders of all priorities by closing the queue context to which they belong.

Addresses

When you send an AOCE message, you must specify the address to which the message is to be delivered. The address can specify an entity (such as a person), an exact location (such as a queue on a specific AppleTalk node), or a group (which must be resolved into individual addresses).

An IPM message can contain two types of addresses: direct addresses and indirect addresses. A direct address specifies the exact location and queue name to which you want the message sent. An indirect address specifies the person or group to which you want the message sent and relies on IPM to determine the actual location and queue name of each addressee. AOCE addressing is described in two sections: "Direct Addressing," beginning on page 7-11, and "Indirect Addressing," beginning on page 7-14.

Report Messages

When you send a message, you can request that the IPM Manager return recipient report messages. You have several options for report messages. You can request that the IPM Manager

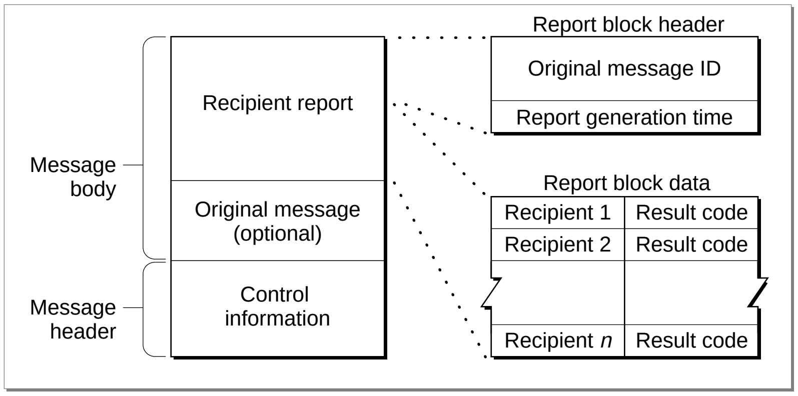
- return report messages when the message is delivered
- return report messages when the message cannot be delivered
- return both delivery and nondelivery reports
- include the original message in the report message
- include the original message only in nondelivery reports
- send a separate report message for each recipient, sending each one as soon as its delivery status is known for that recipient
- wait until the delivery status of the message is known for all recipients and then send a single summary report

Figure 7-4 illustrates the contents of a report message. Note that Figure 7-4 does not show the size or true sequence of fields in the report message. You must use IPM Manager functions to read report message information.

The report message contains a recipient report block, which includes a header and report data. The header, an `IPMReportBlockHeader` structure, includes the message ID of the original message and the time that the IPM Manager generated the report. The report data, an `OCERecipientReport` structure, indicates the outcome of the delivery to each recipient to which the report applies.

Because reports are messages, they are delivered to queues just as all messages are. Report messages are always delivered to the reply queue specified in the original message. If no reply queue was specified in the original message, then the IPM Manager does not issue report messages. When you send a message, you have the option of specifying whether you want the IPM Manager to issue delivery and nondelivery reports.

Figure 7-4 An IPM report message



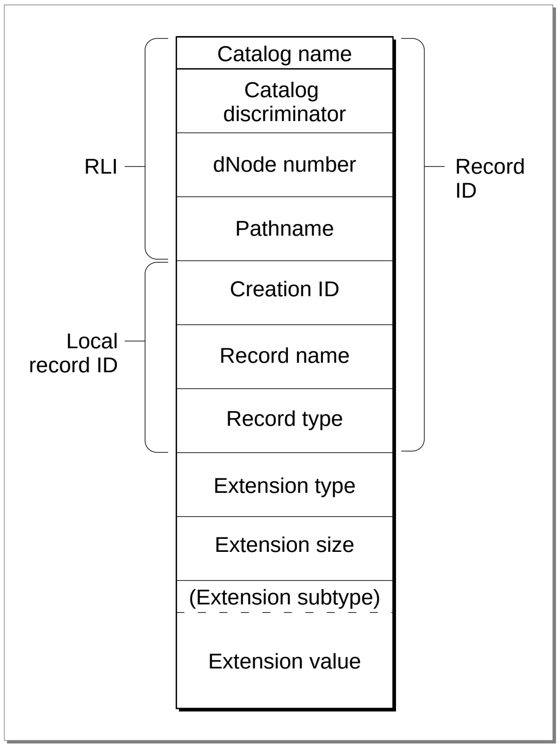
For more information on how to read a report, see the descriptions of the `IPMReportBlockHeader` structure on page 7-33 and the `OCERecipientReport` structure on page 7-33.

Addressing IPM Messages

The IPM Manager uses a single data type, the `OCERecipient` structure, to specify any type of address. Figure 7-5 shows the components of an `OCERecipient` structure. The `OCERecipient` structure has three parts: record location information (RLI), a local record identifier, and an extension. The record location information and local record identifier make up a record ID. Which of these parts are used in a specific address depends on the type of address, as described in the following sections. The `OCERecipient` structure is defined on page 7-24. For more information on the

`RecordID`, `LocalRecordID`, and `RLI` structures, see the chapter "AOCE Utilities" in this book.

Figure 7-5 Contents of an `OCERecipient` structure



Direct Addressing

In direct addressing, the `OCERecipient` structure specifies the location of the recipient and the queue to which you want the message sent. (All AOCE messages are delivered to a specific queue at a specific location.) This information is contained in the extension part of the `OCERecipient` structure.

Apple Computer, Inc., has defined address formats for its built-in transport media, which are described in the following sections. Personal and Server MSAMs allow the transport address space to be extended, and each transport medium has a unique set of addresses. Generally, the record location information (RLI) in the `RecordID` field is used for routing, the name and type are used for display, and the extension contains the native transport address as a displayable `RString`. The list of accessible RLIs is available via the `DirGetExtendedDirectoriesInfo` function, which is defined by the Catalog Manager.

The AOCE software defines two types of direct addresses: the AppleTalk type and the telephone type, described in the following two sections.

AppleTalk Direct Addressing

You can use AppleTalk direct addressing to specify the location on an AppleTalk internet to which a message should be delivered. You can use the Name Binding Protocol (NBP) AppleTalk routines to obtain the addresses of entities on an AppleTalk network. For more information on NBP and AppleTalk networking, see *Inside Macintosh: Networking.*

The IPM Manager recognizes an AppleTalk direct address by the value `'alan'` in the `extensionType` field of the `OCERecipient` structure. In this case, the extension portion of the `OCERecipient` structure contains the entire address; the IPM Manager ignores the record ID portion of the structure. The `extensionValue` field of the `OCERecipient` structure is defined as follows:

```
Str32    objectName
Str32    typeName
Str32    zoneName
Str32    queueName
```

All four of the fields are required, and all are packed. The first three fields are in the exact format used by the NBP `EntityName` structure. As is usual for AppleTalk, you can specify a zero-length string or the wildcard character `*` to indicate the local zone.

You must fill in the `queueName` field with the name of the specific queue to which the message is to be delivered. The messaging applications on both the sending and receiving computers have to open input queues and must somehow exchange queue names. You have to determine the protocol for achieving this yourself. The easiest way to know the recipient's queue name is for your application to use the same queue name always. If you need to send messages to multiple queues or have other reasons to allow more than one possible queue name, you have to implement your own process for determining which queues are available and what their names are.

Telephone Direct Addressing

You can use telephone direct addressing to specify an address for use by PowerTalk Direct Dialup. You must specify a telephone number and the recipient queue name. The IPM Manager delivers the message to the queue with the specified name on the node that is connected to a modem at the specified telephone number. To receive and route the message correctly, the receiving computer must have Direct Dialup installed and the modem set to answer the telephone.

Note
The telephone direct addressing type of `OCERecipient` structure described here is created by the Direct Dialup template when the user adds a Direct Dialup mail address to an information card. You can use the information in this section to create your own Direct Dialup addresses to use with messaging applications. The IPM Manager provides no facilities for using communications software other than Direct Dialup to send messages over telephone lines. ◆

The IPM Manager recognizes a telephone direct address by the value `'aphn'` in the `extensionType` field of the `OCERecipient` structure.

You should use the string "Direct Dialup" for the catalog name field of the `OCERecipient` structure. This is the name of the personal catalog used by the Direct Dialup software for setup information. The Direct Dialup catalog contains access numbers for local calls (such as 9, used to obtain an outside line in some telephone systems), long distance calls (such as 8 to obtain a long-distance outside line), and international calls (when calling from the United States, this is generally 011, the international access code) and can specify a credit card number to be used. The IPM Manager ignores the other fields in the record ID portion of the `OCERecipient` structure.

When you use telephone direct addressing, the `extensionType` field must contain the value `'aphn'` and the `extensionValue` field is defined as follows:

```
RString     phoneNumber /* telephone number */
RString     modemType   /* reserved */
Str32       queueName   /* recipient's queue name */
```

All three fields are required. The `phoneNumber` and `modemType` fields must be padded to an even number of bytes, and all fields must be packed.

The `phoneNumber` field is composed of several subfields. Each subfield must be packed and padded to an even number of bytes.

```
short    subType;
RString  countryCode;
RString  areaCode;
RString  phone;
RString  postFix;
RString  nonHandyDialString;
```

Field descriptions

`subType`	A byte that specifies whether the Direct Dialup software should use the information in the Direct Dialup setup catalog when it forms the dialing string. If you specify the value `kOCEUseHandyDial` for this field, the Direct Dialup software uses the Direct Dialup catalog to obtain special access numbers and optionally a charge card number. If you specify `kOCEDontUseHandyDial` for this field, the Direct Dialup software uses only the exact dialing string you specify in the `nonHandyDialString` field and ignores the other subfields.
`countryCode`	The ASCII value of the country code needed to dial an international number. For example, the country code for the United Kingdom is ASCII 44. For long-distance calls from and within North America, use the long distance prefix, ASCII 1.
`areaCode`	The ASCII value of the US area code or, for international calls, the city code.

`phone`	The telephone number including any other special modem control characters you may need. For example, you could include the "," character as one of the characters in the `phone` string to cause the modem to pause briefly before dialing the rest of a number.
`postFix`	Reserved. You must specify an `RString` structure of zero length and an empty data string (`""`).
`nonHandyDialString`	The dialing string used by the Direct Dialup software when you set the `subType` field to `kOCEDontUseHandyDial`. When this is the case all of the other fields of the extension value are ignored when the dialing string is formed. If the `subType` field has a value of `kOCEUseHandyDial`, then Direct Dialup ignores this field.

The `modemType` field of the extension value is reserved and must be set to an empty `RString`; that is, an `RString` structure with a length of `0` and an empty data string (`""`).

You must fill in the `queueName` field of the extension value with the name of the specific queue to which the message is to be delivered. The messaging applications on both the sending and receiving computers have to open input queues and must somehow exchange queue names. You have to determine the protocol for achieving this yourself. The easiest way to know the recipient's queue name is for your application to always use the same queue name. If you need to send messages to multiple queues or have other reasons to allow more than one possible queue name, you have to implement your own process for determining which queues are available and what their names are.

Indirect Addressing

You can use indirect addressing when you want to specify the entity to which a message should go, instead of the exact location and queue name to which the message should be delivered. In indirect addressing you specify a record—and optionally a specific attribute within the record—that contains the location and queue information that the IPM Manager needs to deliver the message. In mail applications, for example, the user typically selects a user record from a catalog or information card as the addressee. The IPM Manager then looks up the address of the recipient in that user record.

To use indirect addressing, fill in the record ID portion of the `OCERecipient` structure with the record ID of the record containing the address and set the `extensionType` field to the value '`entn`'. Extensions of type '`entn`' include a subtype field, which can have the following values:

```
enum {
   kOCEAddrXtn= 'addr', /* reserved */
   kOCEQnamXtn= 'qnam', /* queue-name form */
   kOCEAttrXtn= 'attr', /* attribute-type form */
   kOCESpAtXtn= 'spat'  /* reserved */
};
```

To specify an indirect address, you must use the attribute-type (`'attr'`) subtype. The queue-name subtype of an `OCERecipient` structure is used for attribute values (see “Queue-Name Format for Attribute Values” on page 7-16). The other two subtypes are reserved for use by the IPM Manager.

Both the attribute-type and queue-name subtypes require the record ID portion of the `OCERecipient` structure to contain a valid reference to a record.

The fields that are required in the record ID portion of the `OCERecipient` structure are as follows:

- If the creation ID value is sufficient to identify the record in the catalog then the `recordName` and `recordType` fields are not required and can be `nil`.
- If the creation ID is not sufficient to specify the record or is `null`, then the `recordName` and `recordType` fields are required.
- If you include both the creation ID and the record name and type, they must specify the same record.

You can use the Catalog Manager functions to create and modify records and record attribute values. See the chapter “Catalog Manager” in this book for more information. For information on record IDs, attributes, attribute values, and the creation ID, see the chapter “AOCE Utilities” in this book.

Attribute-Type Indirect Addressing

You use attribute-type indirect addressing when you want to specify the entity that is to receive a message rather than the specific location and queue to which a message is to be delivered. The IPM Manager obtains the location and queue name to which the message is to be delivered from an attribute in the record you specify. If you are specifying a standard AOCE user record or group record into which the system administrator placed messaging addresses, then the IPM Manager creates the attribute containing the address, and you do not have to be concerned with the format of the attribute value. If, however, you want to create your own record or attribute type and place addresses in it yourself, then you need to be familiar with address formats for attributes, discussed in the following section, “Queue-Name Format for Attribute Values.”

The simplest form of an attribute-type `OCERecipient` structure has an extension type of `'entn'`, an extension size of 0, and no extension value. In this case, the IPM Manager uses the preferred messaging queue as specified in the default messaging attribute in the record. The preferred messaging queue is created and designated by the catalog administrator.

To specify an attribute type, use an extension type of `'entn'` and a subtype of `'attr'`. The extension value is defined as follows:

```
OSType         'attr'
AttributeType  attributeName
```

The `AttributeType` structure is defined as follows:

```
struct AttributeType {
   RStringHeader
   Byte body[kAttributeTypeMaxBytes];
};
```

The `attributeName` field must be packed and padded to an even number of bytes. The `AttributeType` structure is equivalent to an `RString` structure that has a length of `kAttributeTypeMaxBytes` bytes. For more information on the `AttributeType` and `RString` structures, see the chapter "AOCE Utilities" in this book.

Setting the subtype to `'attr'` and the `body` field of the `AttributeType` structure to the value `kPrefMsgQAttrTypeBody` has the same effect as leaving out the extension value entirely: the IPM Manager uses the preferred messaging queue in the record as the address to which to deliver the message.

If you specify another attribute type, then the IPM Manager looks for the address in that attribute type. If there is more than one attribute value in the record with the attribute type you specify, the IPM Manager chooses one of the values. The method that the IPM Manager uses to decide which attribute value to use is private. Therefore, you should use a multivalued attribute type to hold an indirect address only when you do not care at which address a recipient receives the message.

Queue-Name Format for Attribute Values

If you want to define your own record type or attribute type to hold addresses for indirect addressing, you must format the attribute value as an `OCERecipient` structure. You use the queue name form of the `OCERecipient` structure for the attribute value. The recipient must have an account on an AOCE messaging server, such as a PowerShare server. The queue name form specifies the messaging server and queue name to which to deliver the message.

In the queue name form of the `OCERecipient` structure, the `extensionType` field has a value of '`entn`', the extension subtype field has a value of '`qnam`', and the extension data is a queue name string. The `extensionValue` portion of the `OCERecipient` structure is defined as follows:

```
OSType          'qnam'
Str32           queueName
```

The record ID portion of the `OCERecipient` structure specifies the catalog and record ID of the catalog record that contains information about the messaging server. (When the system administrator installs a messaging server, the setup software creates a catalog record containing information about the messaging server.)

As with other AOCE addressing formats that require the name of a queue, you must implement your own method for obtaining the queue name because the AOCE toolbox does not provide you with a mechanism for doing so.

Here is one possible procedure for indirect addressing using queue name attribute values:

1. Create your own new record type, or create a new attribute for an existing record type.
2. Log on to the messaging server as an administrator and create a queue with the name you want to use. You use the `IPMCreateQueue` function (page 7-69) for this purpose.
3. Put the name and location of the queue you just created into the new attribute in a queue-name-format `OCERecipient` structure.
4. Once you have created the queue and you have placed the queue name and location information into an attribute, then both ends of your connection can obtain the queue name from the record. Both the recipient and the sender of the message must know before the message is sent which record and attribute in the catalog contains the queue name.

Using the IPM Manager

This section describes how to create messages, create and manage message queues, and read messages.

Determining Whether the Collaboration Toolbox is Available

Before calling any of the Interprogram Messaging Manager functions, you 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" of *Inside Macintosh: Operating System Utilities*.

If you want to be informed when the Interprogram Messaging Manager starts up or shuts down, you can install an entry in the AppleTalk Transition Queue (ATQ). Then the AppleTalk LAP 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 of *Inside Macintosh: Networking*.

Determining the Version of the Collaboration Toolbox

To determine the version of the Collaboration Toolbox 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 you are using the Collaboration toolbox on a computer that has a PowerShare server, the function returns the version number of the server in the high-order word of the `response` parameter. If the Collaboration Toolbox or server is not present and available, the `Gestalt` function returns 0 for the relevant version number. You can use the constant `gestaltOCETB` for AOCE Collaboration Toolbox version 1.0.

Note that the version number of the Collaboration toolbox is not necessarily the same as that returned by the `IPMReadHeader` function (page 7-89) for the IPM Manager. The `IPMReadHeader` function returns a version number in the `version` field of the `IPMFixedHdrInfo` structure (page 7-38).

Error Handling

If the ASDSP connection between the Collaboration toolbox and the server shuts down for any reason, the next IPM Manager function you call that requires communications with the server fails with the result code `kOCEConnectionClosed`. To correct this condition, call the `IPMCloseQueue` function (page 7-76) to close the messaging queue and then call the `IPMOpenQueue` function (page 7-72) to reopen the queue.

If either end of the IPM connection crashes during message transmission, the IPM Manager might send a duplicate copy of a message that was already successfully delivered. Although such an occurrence is very rare, your application should be capable of handling the receipt of duplicate messages.

Creating a Message

A message is created in three steps:

1. Initiate the message-creation process.
2. Add information to the message.
3. End the process.

Initiating the Message-Creation Process

Before you start to create a message, you must decide whether you intend to send the message, save it to disk, or nest it in another message. These processes are independent of one another. If you want to both send a message and save the same message to disk, for example, you must create the message twice.

- To begin the process of creating a new message to be sent to a recipient, call the `IPMNewMsg` function (page 7-43).
- To start a new message to be saved to disk, call the `IPMNewHFSMsg` function (page 7-47).
- To start a new nested message, call the `IPMNewNestedMsgBlock` function (page 7-56).

You provide each of these functions with information for the message header and an authentication identity of the creator of the message. You can specify the reply message queue and one recipient message queue at this time, or you can add this address information later, as described in the following section. Each of the new-message functions returns a message reference number that you must use when you call other functions to build the message.

Adding Information to the Message

Once you have started the message, you can add information to the message. You can call the `IPMAddRecipient` (page 7-50) and `IPMAddReplyQueue` (page 7-52) functions at any time during the message-creation process to add recipients and a reply queue to the message header. To add a new message block, you first call either the `IPMNewBlock` function (page 7-53) to start a new message block, or the `IPMNewNestedMsgBlock` function to start a new nested-message block. You then call the `IPMWriteMsg` function (page 7-61) to add data to a message block. You can also use the `IPMNestMsg` function (page 7-59) to add an existing message as a message block. You can't modify such a nested message. You can add as many message blocks and nested messages as you wish to a message.

Note
Although the IPM Manager allows you to add any number of nested-message blocks at the same nesting level in a message, the messaging service access module (MSAM) interface supports only one nested-message block at a given nesting level. Therefore, if you want your message to be compatible with MSAMs, you must not add more than one nested-message block at a given level of nesting. You can, however, nest a message within another nested message to as many nesting levels as disk and memory resources allow. ◆

The `IPMWriteMsg` function adds data at a specific offset in a message. You can specify an offset from the start of the currently open message block, from the start of the message, or from the end of the last byte written. A message block can be any length. Each time you call the `IPMNewBlock` function or the `IPMNewNestedMsgBlock` function, the IPM Manager closes the current message block and starts a new message block, putting the offset to the beginning of the new block into the message header. Therefore, once you start a new message block, you cannot extend the length of any message blocks you added earlier. You can write over the data in a block you wrote earlier, but you can't extend the block.

If you call the `IPMNewNestedMsgBlock` function to add a nested-message block to a message, each subsequent call to the `IPMNewBlock` or `IPMNewNestedMsgBlock` functions adds another block to the nested message, not a new block to the enclosing message. Once you have started a nested message, you must call the `IPMEndMsg` function (page 7-65) to complete the nested message before you can add any more information to the enclosing message. After you call the `IPMEndMsg` function to end the nested message, you cannot add any recipients or blocks to the nested message.

Ending a Message

When you are finished adding address information, blocks, and nested messages to your message, you call the `IPMEndMsg` function. This function sends the message, saves it to disk, or ends a nested message, depending on which function you used to start the message. You can also choose to add a digital signature to the message at this time and you can request delivery and nondelivery reports.

Creating and Managing Message Queues

The IPM Manager provides functions to perform the following tasks:

- create a new physical queue
- open a queue context
- open a physical queue to establish a virtual queue
- change the default message filter for a virtual queue
- enumerate the messages in a queue
- close a queue context
- close a virtual queue
- delete a physical queue

Creating and Opening a Queue

Before another client of IPM can send messages to your application or process, you must establish the input messaging queue to which the messages will be sent and from which you can read them. You can use the default messaging queue created by the PowerShare system administrator for the user as described in “Attribute-Type Indirect Addressing” on page 7-15.

The administrator of a PowerShare messaging server can create any number of queues on the server computer. Each such queue has a creator (the administrator who created the queue) and an owner, assigned by the administrator. The owner can open a queue and the administrator can delete a queue. An administrator typically creates a queue for each user who has an account on the server.

However, if you want to create and maintain your own messaging queues, you must use the functions described in “Managing Message Queues” on page 7-68.

To establish a messaging queue, follow these steps:

1. Call the `IPMCreateQueue` function to create a new physical queue. When you call the `IPMCreateQueue` function (page 7-69), the IPM Manager sets up a new physical input queue with the name and address you specify. Other users of IPM can send messages to that queue (assuming they know its name and address) at any time.

Note
You must be authenticated as the administrator to add a queue to a PowerShare server. ◆

2. Call the `IPMOpenContext` function to create a new queue context. A queue context links together virtual queues so that, by closing the context, you can simultaneously close all of the queues associated with that context. You use the `IPMOpenContext` function (page 7-70) to create a new context and the `IPMCloseContext` function (page 7-77) to close one. The `IPMOpenContext` function returns a context reference number that you use when you call the `IPMOpenQueue` function to open a new virtual queue.
3. Call the `IPMOpenQueue` function to establish a new virtual queue. Whereas the `IPMCreateQueue` function creates a physical message queue, the `IPMOpenQueue` function (page 7-72) opens the physical queue to establish a virtual queue (see "Message Queues" on page 7-8 for a discussion of physical and virtual message queues). You cannot read messages from a queue until you open it. When you call the `IPMOpenQueue` function, you must specify the queue context to which the new virtual queue will belong. You can call the `IPMOpenQueue` function any number of times to establish distinct virtual queues associated with the same physical input queue. Each time you call this function, the IPM Manager returns a unique queue reference number.

Specifying a Queue Filter and Enumerating a Queue

When you call the `IPMOpenQueue` function to establish a virtual queue, you can specify a default message filter for that virtual queue. You can filter messages by priority, message type, or other attributes, as described in "Filter Structures" on page 7-34.

For example, you can open an input queue three times to create three virtual queues, each with its own filter: one that passes only high-priority messages, one that passes only messages specifically intended for your application, and one that passes all messages in the physical input queue. You can use the `IPMChangeQueueFilter` function (page 7-74) to change the default message filter for a specific virtual queue.

When you call the `IPMEnumerateQueue` function (page 7-80), you specify a queue reference number and you can specify a queue filter. The IPM Manager uses the message filter to determine which messages in the physical queue to list. If you do not provide a message filter with the `IPMEnumerateQueue` function, the function uses the default filter for that virtual queue.

Closing a Queue

You can close an individual virtual queue or you can close a queue context to simultaneously close all of the virtual queues associated with that context. When you open a message, you specify the reference number for an open virtual queue. This virtual queue must belong to the physical queue that actually contains the message and its filter must pass the specific message you wish to open. When you call the `IPMCloseQueue` function (page 7-76) to close a virtual queue, the IPM Manager closes all of the messages opened using that virtual queue's reference number and removes the virtual queue from its context. When you call the `IPMCloseContext` function (page 7-77) to close a context, the IPM Manager closes all of the messages opened for all the virtual queues associated with that context before it closes the virtual queues and removes the context.

Call the `IPMDeleteQueue` function (page 7-78) to delete a physical queue that you own. Before you delete a physical queue, you must close all of the virtual queues that belong to that physical queue.

Reading Messages

To read a message, follow these steps:

1. Enumerate the queue or determine the location of the message on disk. Use the `IPMEnumerateQueue` function (page 7-80) to list the messages in a virtual queue; that is, the messages that meet the filter criteria for the queue. If you wish, you can specify a filter that is in effect only for a single execution of the function; otherwise, the function uses the current filter for the virtual queue. In addition to the sequence number of each message, the `IPMEnumerateQueue` function provides information about the message such as the message length and priority.

   A queue can contain any number of messages. The IPM Manager assigns a sequence number to each message when it adds the message to the physical queue. The IPM Manager uses a monotonically increasing series of sequence numbers and does not reuse a sequence number when a message is deleted from the queue. Therefore, when you request a list of all the messages in the queue, some sequence numbers might be missing, but the message with the highest sequence number is always the last one added to the queue.

   Use File Manager or Standard File Package routines to locate a message on disk. The File Manager and Standard File Package are described in *Inside Macintosh: Files*.

2. Open the message. Use the `IPMOpenMsg` function (page 7-82) to open a message in an input queue or the `IPMOpenHFSMsg` function (page 7-84) to open a message that has been saved in a file on disk. These functions return a message reference number that you must provide to the various message-reading functions.

   If a message contains a nested-message block, you can use the `IPMOpenBlockAsMsg` function (page 7-86) to open that block as a message. You must open the containing message and determine the offsets of the nested-message block before you can open a nested message. You use the `IPMGetBlkIndex` function (page 7-96) to get the index numbers and block types of the blocks in a message.

3. Read the message header. The IPM Manager reads certain fields of the headers of messages in an input queue and saves this information in local memory. You can use the `IPMGetMsgInfo` function (page 7-87) to read this information. The `IPMGetMsgInfo` function returns the same information about a message as that returned by the `IPMEnumerateQueue` function. To get more information about a message or to read header information from a message on disk or a nested message, use the `IPMReadHeader` function (page 7-89).

   The creator of a message adds one or more recipients to the message header. Some or all of these recipients might be group addresses or references to catalog records that the IPM Manager must resolve before delivering the message. The `IPMReadRecipient` function (page 7-92) returns only the original list of recipients.

4. Call the `IPMGetBlkIndex` function (page 7-96) to get the index numbers and block types of the blocks in the message. If you are interested only in blocks of a certain type, such as nested-message blocks, you can use this function to list only those blocks.

5. Use the `IPMReadMsg` function (page 7-98) to read any message block other than a nested-message block.

   Call the `IPMOpenBlockAsMsg` function to open a nested-message block as a message and then use the other functions in this section to read it as you would read any other message. Before you use this function, you must open the containing message (which can also be a nested message) and you must know the index number of the nested-message block within the containing message. A nested message has a creator type of `kIPMSignature` and a block type of `kIPMEnclosedMsgType`.

   If the message includes a digital-signature block, you can use the `IPMVerifySignature` function (page 7-102) to verify the signature.

6. When you have finished reading the message, call the `IPMCloseMsg` function (page 7-104) to close the message and release the memory the IPM Manager reserved for the message when you opened it. Closing a message does not automatically close any nested messages that you have opened with the `IPMOpenBlockAsMsg` function; you must call the `IPMCloseMsg` function once for every nested message you open. You can also close messages by closing the message queue or the queue context to which that message belongs.

IPM Manager Reference

This section describes the data types and routines provided by the IPM Manager.

Data Types

The IPM Manager routines use the data types described in this section. Included are structures for message addressing, message and block types, delivery notification, filter structures, message information structures, header information structures, sender structures, and interprogram messaging parameter blocks.

Message Addressing Structures

You must use the `OCERecipient` structure to specify a message address. This section also shows some structures you can use for extensions to `OCERecipient` structures. See "Addressing IPM Messages," beginning on page 7-10 for more information about addressing.

OCERecipient

The `OCERecipient` structure is defined as a `DSSpec` data type.

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

typedef struct DSSpec DSSpec;
typedef DSSpec OCERecipient;
```

The `OCERecipient` structure can specify a specific attribute in a specific record in a catalog from which the IPM Manager reads the recipient address, or it can hold the actual queue address. The various forms of the `OCERecipient` structure are described in "Addressing IPM Messages," beginning on page 7-10.

All of the components of the DSSpec data type are defined in the chapter "OCE Utilities" in this book. Figure 7-5 on page 7-11 illustrates the contents of an OCERecipient structure. Note that this figure does not show the true size or location of the fields in an OCERecipient structure, and that the actual structure contains packed fields. You must use the utility routines provided by the IPM Manager to create and read these structures. The utility routines are described in "Utility Functions," beginning on page 7-107.

OCEPackedRecipient

The IPM Manager often uses a packed form of the OCERecipient structure, defined by the OCEPackedRecipient data type.

```
# define OCEPackedRecipientHeader\
   unsigned short    dataLength;

struct ProtoOCEPackedRecipient{
   OCEPackedRecipientHeader;
};

typedef struct ProtoOCEPackedRecipient ProtoOCEPackedRecipient;

# define kOCEPackedRecipientMAXBYTES\
         (4096 - sizeof(ProtoOCEPackedRecipient))

struct OCEPackedRecipient {
   OCEPackedRecipientHeader
   Byte              data[kOCEPackedRecipientMaxBytes];
};

typedef struct OCEPackedRecipient OCEPackedRecipient;
```

The dataLength field at the beginning of the structure specifies the length of the data field that follows. The data field of the OCEPackedRecipient structure contains an OCERecipient structure in packed format. Use the utility routines provided by the IPM Manager to pack and unpack OCERecipient structures.

IPMEntityNameExtension

You can use the following data type when creating an extension to an `OCERecipient` structure:

```
struct IPMEntityNameExtension {
   OSType subExtensionType;
   union {
      IPMEntnSpecificAttributeExtension specificAttribute;
      IPMEntnAttributeExtension         attribute;
      IPMEntnQueueExtension             queue;
   } u;
};
```

The specific attribute type is reserved for use by the IPM Manager.

IPMEntnAttributeExtension

The attribute type is defined by the `IPMEntnAttributeExtension` structure.

```
struct IPMEntnAttributeExtension {     /* kOCEAttrXtn */
   AttributeType attributeName;
};
```

IPMEntnQueueExtension

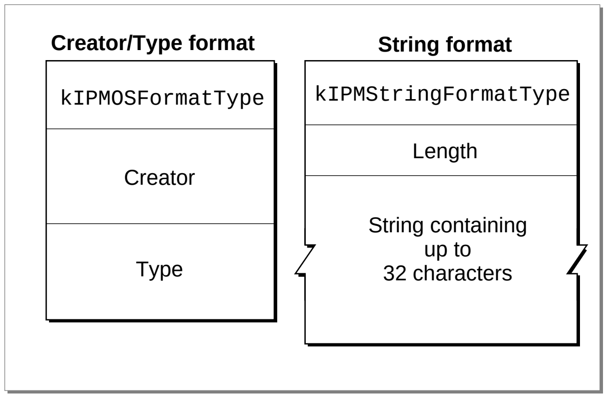
The queue type is defined by the `IPMEntnQueueExtension` data structure.

```
struct IPMEntnQueueExtension {
   Str32 queueName;
};
```

Message and Block Types

Each IPM message has an associated message type. Each block in a message has a block type. A message type can have either of two formats: the creator/type format contains a creator field and a type field; the string format contains a length field and a string. A block type always has the creator/type format. As illustrated in Figure 7-6, the first field in a message type structure is a 2-byte tag that specifies the format of the structure. The block type structure does not include this tag.

Figure 7-6 The two forms of the message type structure



IMPORTANT
Apple Computer, Inc., reserves all message type values and all block type values that consist entirely of lowercase letters. ▲

OCECreatorType

The block type and the creator/type portion of a message type are defined by the `OCECreatorType` data type.

```
struct OCECreatorType {
   OSType   msgCreator;
   OSType   msgType;
};
```

Field descriptions

`msgCreator`	The creator of the message or block. You can specify any four-character value in this field; usually it is the signature of your application. For example, a message or block created by the IPM Manager has a creator type of `kIPMSignature`.
`msgType`	The type of the message or block. For example, an enclosed message block has a block type of `kIPMEnclosedMsgType`. You can define your own four-character block types to serve your own purposes. Apple Computer, Inc., reserves all block types consisting entirely of lowercase letters.

IPMMsgType

The message type structure is defined by the `IPMMsgType` data type.

```
/* values of IPMMsgFormat */
enum {
   kIPMOSFormatType = 1,
   kIPMStringFormatType = 2
};

typedef Str32 IPMStringMsgType;

struct IPMMsgType {
   IPMMsgFormat         format;     /* IPMMsgFormat */
   union{
      OCECreatorType    msgOSType;
      IPMStringMsgType  msgStrType;
   }theType;
};

typedef struct IPMMsgType IPMMsgType;
```

IPMBlockType

The block type structure is defined by the `IPMBlockType` data type.

```
typedef OCECreatorType IPMBlockType;
```

Delivery Notification

The IPM Manager uses a delivery notification flag byte in the message header to determine when to generate recipient report messages and whether to include the original message in any report messages that are returned by the recipients. Report messages include a header (the `IPMReportBlockHeader` structure on page 7-33) and an array of delivery results (the `OCERecipientReport` structure on page 7-33). Report messages are described in “Report Messages” on page 7-9.

Nondelivery Codes

The nondelivery result codes that can be returned by an MSAM or the IPM Manager in a recipient report message are shown here. A personal MSAM can define its own result codes in addition to the ones listed here. (If a server MSAM returns a nonstandard result code, the IPM Manager is unable to convert it to a string meaningful to the user.)

```
enum {
   kIPMNoSuchRecipient                 =   0x0001,
   kIPMRecipientMalformed              =   0x0002,
   kIPMRecipientAmbiguous              =   0x0003,
   kIPMRecipientAccessDenied           =   0x0004,
   kIPMGroupExpansionProblem           =   0x0005,
   kIPMMsgUnreadable                   =   0x0006,
   kIPMMsgExpired                      =   0x0007,
   kIPMMsgNoTranslatableContent        =   0x0008,
   kIPMRecipientReqStdCont             =   0x0009,
   kIPMRecipientReqSnapShot            =   0x000A,
   kIPMNoTransferDiskFull              =   0x000B,
   kIPMNoTransferMsgRejectedbyDest     =   0x000C,
   kIPMNoTransferMsgTooLarge           =   0x000D
}
```

Constant descriptions

`kIPMNoSuchRecipient`
: The IPM Manager or MSAM has determined that the specified recipient does not exist. For example, the recipient might have no record in the catalog (and therefore no account on the mail server) or have no account on the MSAM's mail or messaging system.

`kIPMRecipientMalformed`
: The recipient address in the message was not formatted correctly. The problem can be any of the following: The name and record creation ID don't match; both the dNode number and pathname are specified in the record location information (RLI) structure; a dialup address is missing a phone number; an NBP address is missing a zone name; the RLI for a catalog is missing a discriminator; the extension value of the `OCERecipient` structure is not properly formed (as determined by the MSAM interpreting the address).

`kIPMRecipientAmbiguous`
: The IPM Manager or MSAM has been unable to resolve, look up, or find the specified recipient. The recipient may exist but has been unavailable (for example, it has an AppleTalk address but has not been logged on to AppleTalk), or there may be duplicate addresses and the IPM Manager or MSAM cannot determine which to use.

`kIPMRecipientAccessDenied`
: In the process of attempting to deliver the message to the specified recipient, access to some critical information was prevented. The address may be valid and the recipient might exist, but the agent responsible for delivering the message doesn't have access to the recipient's record.

`kIPMGroupExpansionProblem`
: The IPM Manager or MSAM was unable to expand a group address fully. Some of the recipients in the group might have received the message.

`kIPMMsgUnreadable`
: The MSAM was unable to read (and thus to translate) a message (the message might be corrupted or the content missing), and therefore the message was never delivered to the specified recipient.

`kIPMMsgExpired`
: The IPM Manager was unable to confirm delivery of this message before the specified expiration time (currently set at 5 days for PowerShare servers, Direct AppleTalk, and server MSAMs). The server makes several attempts to deliver a message before the message delivery time expires. This result code does not necessarily mean that all the attempts at delivery failed— it means that the server has not been able to determine the success or failure of any of the previous attempts to deliver the message and will make no further attempts.

`kIPMMsgNoTranslatableContent`
: The message is missing a piece of information that is considered critical for its delivery. For example, the message might be missing a subject or a type of content required by the MSAM.

`kIPMRecipientReqStdCont`
: The MSAM cannot deliver messages that don't contain a standard-interchange-format block, and such a block was not present.

`kIPMRecipientReqSnapShot`
: The MSAM required the message to contain a standard image format block (or *snapshot*) in order to deliver it, and such a block was not present.

`kIPMNoTransferDiskFull`
: The recipient could not receive the message because there was insufficient room on the disk to hold it. The recipient might be a user's computer in the case of Direct AppleTalk or a server in the case of an MSAM. If a PowerShare disk is full, the IPM Manager periodically makes new attempts to send the message.

`kIPMNoTransferMsgRejectedbyDest`
: The destination system refused delivery without specifying a reason.

`kIPMNoTransferMsgTooLarge`
: The destination system has a limit to the size of message it accepts, and this message exceeded that limit.

IPMNotificationType

The IPM delivery notification setting is specified by the `IPMNotificationType` data type.

```
typedef Byte IPMNotificationType;
```

The bits in the notification byte are defined as follows:

```
enum {
   kIPMDeliveryNotificationBit      = 0,
   kIPMNonDeliveryNotificationBit   = 1,
   kIPMEncloseOriginalBit           = 2,
   kIPMSummaryReportBit             = 3,
   kIPMOriginalOnlyOnErrorBit       = 4
};
```

You can use a combination of the following values to set the flags in the `IPMNotificationType` data type:

```
enum {
   kIPMNoNotificationMask           = 0x00,
   kIPMDeliveryNotificationMask     = 1<<kIPMDeliveryNotificationBit,
   kIPMNonDeliveryNotificationMask  = 1<<kIPMNonDeliveryNotificationBit,
   kIPMDontEncloseOriginalMask      = 0x00,
   kIPMEncloseOriginalMask          = 1<<kIPMEncloseOriginalBit,
   kIPMImmediateReportMask          = 0x00,
   kIPMSummaryReportMask            = 1<<kIPMSummaryReportBit,
   kIPMOriginalOnlyOnErrorMask      = 1<<kIPMOriginalOnlyOnErrorBit,
   kIPMEncloseOriginalOnErrorMask   = 
                          (kIPMOriginalOnlyOnErrorMask|kIPMEncloseOriginalMask)
};
```

Constant descriptions

`kIPMNoNotificationMask`
: Do not deliver any report messages. This setting is overridden when combined with any setting that requests reports.

`kIPMDeliveryNotificationMask`
: Generate a report message when the message arrives at the recipient queue.

`kIPMNonDeliveryNotificationMask`
: Generate a report message if the IPM Manager cannot deliver the message to a recipient.

`kIPMDontEncloseOriginalMask`
: Don't enclose the original message in the report message. This is the default setting for this feature; this setting is overridden by the `kIPMEncloseOriginalMask` setting.

`kIPMEncloseOriginalMask`
: Enclose the original message in a report message. This value must be combined with the `kIPMSummaryReportMask` value.

`kIPMImmediateReportMask`
: Generate a report message for each recipient as soon as there is any information to report. This is the default setting for this feature; this setting is overridden by the `kIPMSummaryReportMask` setting.

`kIPMSummaryReportMask`
: Return a single report message for all recipients.

`kIPMOriginalOnlyOnErrorMask`
: Return the original message only in nondelivery reports. For this setting to have an effect, it must be combined with the `kIPMEncloseOriginalMask` value. The `kIPMEncloseOriginalOnErrorMask` value provides this combination.

`kIPMEncloseOriginalOnErrorMask`
: A combination of the `kIPMEncloseOriginalMask` and `kIPMOriginalOnlyOnErrorMask` values, resulting in the original message being included only in nondelivery reports.

The bit `kIPMSummaryReportBit` in the `IPMNotificationType` byte determines whether the report messages that the sending application receives contain information about a single recipient or all of the recipients of the message. If the bit `kIPMSummaryReportBit` is not set, the IPM Manager returns a report message about each recipient as soon as it is generated. If that bit is set, the IPM Manager creates a single report message that summarizes the requested delivery notification for all of the recipients.

IPMMsgID

The message ID is a unique identifier of the message you sent. The message ID is returned by the `IPMEndMsg` function (page 7-65).

```
struct IPMMsgID {
   unsigned long id[4];
};
```

IPMReportBlockHeader

A recipient report message (message creator `kIPMSignature`, message type `kIPMReportInfo`) includes a report block (which also has a creator of `kIPMSignature` and a type of `kIPMReportInfo`). The report block starts with a header, followed by the report data (see Figure 7-4 on page 7-10). The report block header is defined by the `IPMReportBlockHeader` data type.

```
struct IPMReportBlockHeader {
   IPMMsgID    msgID;         /* message ID of the original */
   UTCTime     creationTime;  /* creation time of the report */
};
```

Field descriptions

`msgID`	The message ID of the message you sent originally. The recipient report message carries information about this message. The message ID is returned by the `IPMEndMsg` function (page 7-65).
`UTCTime`	The time at which the report was generated. The `UTCTime` data type is an unsigned long containing Greenwich Mean Time in seconds since 00:00 hours, January 1, 1904.

OCERecipientReport

A recipient report message (message creator `kIPMSignature`, message type `kIPMReportInfo`) includes a report block (which also has a creator of `kIPMSignature` and a type of `kIPMReportInfo`). The report block starts with a header, followed by the report data (see Figure 7-4 on page 7-10). The report data consists of an array of recipients and delivery results defined by the `OCERecipientReport` data type.

```
struct OCERecipientReport {
   unsigned short rcptIndex;  /* index of recipient in
                                    original message */
   OSErr          result;     /* result of sending letter to
                                    this recipient */
};
```

Field descriptions

`rcptIndex`	The index number of the recipient in the header of the original message. In the case of group addresses, the delivery report tells you only that the group address was expanded; you don't receive information on delivery to individual members of a group.
`result`	The result of the attempt to deliver the message to this recipient. The standard values returned in this field are shown on page 7-29; in addition, each personal MSAM can define its own result codes.

To calculate the number of recipients in a report, divide the size of the block (minus the header size) by the size of an `OCERecipientReport` structure.

```
numRecipients = (pmPB.readMsgPB.actualCount
 - sizeof (IPMReportBlockHeader)) / sizeof (OCERecipientReport);
```

Filter Structures

When you open a message queue or enumerate the messages in the queue, you can apply a filter to the queue so that the IPM Manager lists only the messages that match your filter criteria.

The IPM Manager defines a queue filter as an array of single filters. It performs an OR operation on all of the single filters you specify for a queue filter. For example, if you set one single filter in the filter array to pass high-priority messages of type `'high'` and another single filter to pass low-priority messages of type `'low '`, the queue filter passes messages of both descriptions. The OR operation is performed on the entire single filters, not on the individual fields in the single filters; thus the filter in this example would not pass a low-priority message of type `'high'`.

This section provides the data structures that define single filters and queue filters.

IPMSingleFilter

The `IPMSingleFilter` data type describes the contents of a single filter. You must pack and word-align each field of the structure before you pass it to an IPM routine.

```
struct IPMSingleFilter{
   IPMPriority    priority;
   Byte           padByte;
   OSType         family;  /* family to which this msg belongs */
   ScriptCode     script;  /* language identifier */
   IPMProcHint    hint;
   IPMMsgType     msgType;
};
```

Field descriptions

`priority` The priority of the message. You can set the priority to any of the following values:

```
kIPMAnyPriority
kIPMNormalPriority
kIPMLowPriority
kIPMHighPriority
```

If you set the filter priority to `kIPMAnyPriority`, the queue does not filter messages according to their priority settings.

`family`	The message family to which the message belongs. You can use the wildcard value `kIPMFamilyWildCard` for all families.
`script`	Reserved.
`hint`	A process hint value. A process hint is a string of up to 32 characters, defined by the creator of the message.
`msgType`	A message type. The message type is assigned by the creator of the message. You can use the wildcard value `kIPMTypeWildCard` for either or both fields of the `IPMMsgType` structure to pass messages with any creator or any type. The `IPMMsgType` data type is defined on page 7-28.

The IPM Manager defines the following message family types:

```
#define kIPMFamilyUnspecified 0             /* any message */

#define kIPMFamilyWildCard    0x3F3F3F3FL    /* '????' */
```

In addition, the AOCE MSAM interface defines the following message family types:

```
#define kMailFamily        'mail'   /* "mail" msgs: content, header, etc */

#define kMailFamilyFile    'file'   /* "direct display" msgs */
```

In addition to the types shown here, Apple Computer reserves for its own use any message family type consisting entirely of lowercase letters.

IPMFilter

A full queue filter is a packed array of single filters. The contents of a filter are shown by the `IPMFilter` data type.

```
struct IPMFilter{
   unsigned short    count;
   IPMSingleFilter   sFilters[1];
};
```

Field descriptions

`count`	The number of single filters in this queue filter.
`sFilters`	An array of single filters.

Message Information Structure

When you call the `IPMEnumerateQueue` function (page 7-80) or the `IPMGetMsgInfo` function (page 7-87), the function returns the information about the message in an message information structure.

IPMMsgInfo

The message information structure is defined by the `IPMMsgInfo` data type.

```
struct IPMMsgInfo{
   IPMSeqNum         sequenceNum;
   unsigned long     userData;
   unsigned short    respIndex;
   Byte              padByte;
   IPMPriority       priority;
   unsigned long     msgSize;
   unsigned short    originalRcptCount;
   unsigned short    reserved;
   UTCTime           creationTime;
   IPMMsgID          msgID;
   OSType            family;  /* family of this msg */
   IPMProcHint       procHint;/* packed and even-length padded */
   IPMMsgType        msgType; /* packed and even-length padded */
};
```

The `IPMEnumerateQueue` function lets you specify whether the returned `IPMMsgInfo` structure includes the `procHint` or `msgType` fields. Because these fields are of variable length, the offset to the `msgType` field depends on the presence and length of the `procHint` field.

Field descriptions

`sequenceNum`	A sequence number that uniquely identifies a particular message in the queue.
`userData`	Reserved.
`respIndex`	Reserved.
`priority`	The priority setting of the message. This field can be set to `kIPMNormalPriority`, `kIPMLowPriority`, or `kIPMHighPriority`.
`msgSize`	The length of the entire message.

`originalRcptCount`	The number of recipients that the sending application originally specified for the message. This value may differ from the actual number of recipients if the message was sent to one or more groups.
`reserved`	Reserved.
`creationTime`	The date and time that the message was created. The `UTCTime` data type is an unsigned long containing Greenwich Mean Time in seconds since 00:00 hours, January 1, 1904.
`msgID`	A unique identifier of the message. The message ID is returned by the `IPMEndMsg` function (page 7-65).
`family`	The message family to which the message belongs. Possible values for this field are shown on page 7-35.
`procHint`	An optional field of varied length. If this field is present, it contains the process hint for the message, which is a Pascal-type string of up to 32 characters, defined by the creator of the message. The information in the field is packed. If the field contains an odd number of bytes (including the length byte), the IPM Manager adds a pad byte following the field. Therefore, the maximum length of this field (including the length byte and the pad byte) is 34 bytes.
`msgType`	An optional parameter that contains the message type of the message. The `IPMMsgType` data type is defined on page 7-28. Like the `procHint` field, the `msgType` field is packed and padded if necessary to contain an even number of bytes.

Header Information Structures

The `IPMReadHeader` function (page 7-89) uses the data structures in this section to return information from a message header.

IPMTOC

When you specify the value `kIPMTOC` for the `fieldSelector` field in the parameter block used by the `IPMReadHeader` function, the function returns an array of TOC information structures—one for each block in the message. The TOC information structure is defined by the `IPMTOC` data type.

```
struct   IPMTOC
{
   IPMBlockType      blockType;
   long              blockOffset;
   unsigned long     blockSize;
   unsigned long     blockRefCon;
};
```

Field descriptions

`blockType`	The creator and type of the block.
`blockOffset`	The offset from the start of the message to the start of the block.
`blockSize`	The size, in bytes, of the block.
`blockRefCon`	The block's reference constant. The application that creates the message specifies this value when it adds the block to the message. The meaning of this reference constant is defined by the application that creates the message.

IPMFixedHdrInfo

When you specify the value `kIPMFixedInfo` for the `fieldSelector` field of the parameter block used by the `IPMReadHeader` function, the function returns information about the message header in a fixed header information structure. The fixed header information structure is defined by the `IPMFixedHdrInfo` data type.

```
struct IPMFixedHdrInfo {
   unsigned short      version;          /* IPM Manager version */
   Boolean             authenticated;    /* was message authenticated? */
   Boolean             signatureEnclosed;/* digital signature enclosed? */
   unsigned long       msgSize;          /* size of message */
   IPMNotificationType notification;     /* notification type requested */
   IPMPriority         priority;         /* message priority */
   unsigned short      blockCount;       /* number of blocks */
   unsigned short      originalRcptCount;/* original number of recipients */
   unsigned long       refCon;           /* application-defined data */
   unsigned short      reserved;         /* reserved */
   UTCTime             creationTime;     /* message creation time */
   IPMMsgID            msgID;            /* message ID */
   OSType              family;           /* family of this msg */
};
```

Field descriptions

`version`	The version number of the IPM Manager that created the message. This is not necessarily the same version number as that returned by the `Gestalt` function for the Collaboration toolbox (see page 7-17).
`authenticated`	A Boolean value that indicates whether the message was authenticated. In the case of a message that passes through more than one store-and-forward server, the IPM Manager sets this field to `true` only if the identities of the original sender and of every server in the routing chain were authenticated.

`signatureEnclosed`	A Boolean value indicating whether the message includes a digital signature. If this field is set to `true`, the message includes a block with a creator of `kIPMSignature` and a type of `kIPMDigitalSignature` containing a digital signature. You can use the `IPMVerifySignature` function (page 7-102) to verify the digital signature.
`msgSize`	The length, in bytes, of the message.
`notification`	The delivery notification requested by the application that sent the message. See "Delivery Notification," beginning on page 7-28, for more information about this value.
`priority`	The priority setting of the message. Values for this field can be `kIPMNormalPriority`, `kIPMLowPriority`, or `kIPMHighPriority`.
`blockCount`	The number of blocks in the message. You can use the `IPMGetBlkIndex` function (page 7-96) to list the creator, type, and position of each block in the message.
`originalRcptCount`	The number of recipients in the recipient list that the sending application originally specified for the message. Because the IPM Manager might have expanded groups in the original recipient list, the number of recipients in the current recipient list might be different from this.
`refCon`	A numeric reference value that the sending application provides for the message when it calls the `IPMNewMsg` function (page 7-43), the `IPMNewHFSMsg` function (page 7-47), or the `IPMNewNestedMsgBlock` function (page 7-56).
`reserved`	Reserved.
`creationTime`	The date and time that the message was created. The `UTCTime` data type is an unsigned long containing Greenwich Mean Time in seconds since 00:00 hours, January 1, 1904.
`msgID`	A unique identifier of the message. The message ID is returned by the `IPMEndMsg` function (page 7-65).
`family`	The family the message belongs to.

Sender Structure

When you create a new message or read a message header, the name of the originator of the message is held in a sender structure, described in this section. In the case of an application-to-application message, the sender would be an application name. In the case of a message or letter sent by a user, the sender might be the user's name or a record ID that identifies the user record for the sender.

IPMSender

The sender structure contains either the sender's name in `RString` format or a catalog record ID that identifies the user record for the sender of the message. The sender structure is defined by the `IPMSender` data type.

```
struct IPMSender {
   IPMSenderTag         sendTag;
   union {
      RString           rString;
      PackedRecordID    rid;
   } theSender;
};

enum {
   kIPMSenderRStringTag,
   kIPMSenderRecordIDTag
};
typedef unsigned short IPMSenderTag;
```

Interprogram Messaging Parameter Block Header

All IPM Manager function declarations include a pointer to a parameter block. Each parameter block begins with the following fields:

```
#define IPMParamHeader     \
   Ptr      qLink;         \
   long     reservedH1;    \
   long     reservedH2;    \
   ProcPtr  ioCompletion;  \
   OSErr    ioResult;      \
   long     saveA5;        \
   short    reqCode;
```

Field descriptions

`qLink`	Reserved.
`reservedH1`	Reserved.
`reservedH2`	Reserved.
`ioCompletion`	A pointer to a completion routine that you provide. If you provide a pointer to a completion routine in this field, the function calls your completion routine when it completes execution. Completion routines are described in"Application-Defined Functions," beginning on page 7-114. Specify `nil` for this parameter if you do not want to supply a completion routine.

`ioResult`	The function result. If you call the function asynchronously, it sets this field to 1 to indicate that the request was queued successfully. The function sets this field to the function result when it completes execution.
`saveA5`	Reserved.
`reqCode`	Reserved.

The individual routine descriptions at the end of this reference contain information about any additional parameters that are specific to the routine.

Asynchronous or Synchronous Operations

You can call the IPM Manager routines either synchronously or asynchronously. If you call the function asynchronously, it returns control to you immediately and completes execution incrementally as it is given time by the system. If you call it synchronously, it completes execution before returning control to you.

IMPORTANT

You must specify asynchronous operation when you call any IPM function at interrupt time. Because a function might not complete successfully, calling it synchronously might cause the computer to hang. ▲

Completion Routines and Polling Options

When you call an IPM function asynchronously, you can specify a completion routine. The IPM Manager calls your completion routine when the function completes execution. If you write you completion routine in Pascal or C, it must take a single argument, which is a pointer to the parameter block.

For example, to declare a completion routine in Pascal, you could use the following statement:

```
PROCEDURE MyCompletionRoutine (paramBlk: Ptr);
```

To declare a completion routine in C, you could use the following statement:

```
pascal void MyCompletionRoutine (Ptr paramBlk);
```

If you write your completion routine in assembly language, you can find a pointer to the parameter block in the A0 register and the function result in the D0 register.

The IPM Manager saves the value of your A5 register at the time you call an IPM function and restores the A5 value before calling your completion routine.

If you do not provide a completion routine, you can poll the `ioResult` field of the parameter block. The IPM Manager sets the value of the `ioResult` field to 1 when you first call a function asynchronously, indicating that the function was successfully queued. When the function completes execution, the IPM Manager changes the `ioResult` value to the actual function result.

IPM Manager Functions

This section describes all of the functions provided by the IPM Manager except for those specifically for use by MSAMs; see the chapter "Messaging Service Access Modules" in *Inside Macintosh: AOCE Service Access Modules* for descriptions of MSAM functions.

In the functions described here, you must completely specify any data structure that you provide to a function unless the description states otherwise.

All of the functions take a pointer to an `IPMParamBlock` parameter block as input. Each function description includes a list of the fields in the parameter block that are used by the function.

Most functions in the IPM API have the following form:

```
pascal OSErr function (IPMParamBlockPtr paramBlock,
                        Boolean async);
```

Some functions can be called only synchronously or only asynchronously; therefore, they do not have the `asyncFlag` parameter. The form of those functions is

```
pascal OSErr function (IPMParamBlockPtr paramBlock);
```

The function returns its result code in the `ioResult` field of the parameter block. When you call a function synchronously, it returns its result both as the function result and in the `ioResult` field of the `MailParamBlockHeader` structure. Note that the function also clears the `ioCompletion` field.

When you call a function asynchronously and the function has successfully queued the request, it returns `noErr` and sets the `ioResult` field to 1. After the call completes, the function sets the `ioResult` field to the actual result and calls your completion routine if you specified one. There is one exception to this behavior: if the IPM Manager is not currently ready to accept a request, it may return `corErr` as the function result. In this case, the `ioResult` field has an indeterminate value and the completion routine is not called.

IMPORTANT

If you choose to poll the `ioResult` field to determine if the request has completed, it is safest to check that its value has changed from 1 to some other value. Although the IPM Manager does not return positive error codes, system utilities may return positive error codes and these may be passed through. ▲

Calling an IPM Function From Assembly Language

You can call a function from assembly language. Listing 7-1 illustrates one way to do this for a function that takes both the parameter block pointer and the `async` flag as parameters. (If a function can be called only synchronously or only asynchronously, the assembly code would not manipulate the `async` value.)

Listing 7-1 Calling an MSAM function from assembly language

```
_oceTBDispatch    OPWORD   $AA5E
   SUBQ     #2,A7                  ; make room for function result
   MOVEA    paramBlock,-(SP)       ; push the param block pointer onto stack
   MOVEQ    asyncFlag, D0          ; move async flag into D0
   MOVE.B   D0,-(SP)               ; push the flag (byte) onto stack
   MOVEQ    #opCode, D0            ; move op code into D0
   MOVE.W   D0,-(SP)               ; place the op code on the stack
   _oceTBDispatch                  ; trap call
   MOVE.W   (SP)+, D0              ; get result code
```

Note
The functions described in the section “Utility Functions,” beginning on page 7-107 use a different assembly-language calling convention, described on page 7-107. ◆

Creating a New Message

This section describes the functions that you use to create a new message and either send it or save it to disk. See “Creating a Message,” beginning on page 7-18 for information about the sequence in which you use these functions to create a message.

IPMNewMsg

The `IPMNewMsg` function starts the process of creating a new message to be sent to a recipient.

```
pascal OSErr IPMNewMsg(IPMParamBlockPtr paramBlock,
                       Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`recipient`	`OCERecipient*`	Pointer to the recipient's queue address.
→	`replyQueue`	`OCERecipient*`	Pointer to the queue address for message replies.
→	`procHint`	`StringPtr`	Pointer to character string for your use.
→	`msgType`	`IPMMsgType*`	Pointer to the message type.
→	`refCon`	`unsigned long`	Reserved for your use.
←	`newMsgRef`	`IPMMsgRef`	Message reference number.
→	`identity`	`AuthIdentity`	Authentication identity.
→	`sender`	`IPMSender*`	Pointer to the sender's name.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`recipient`	A pointer to an `OCERecipient` structure that either specifies a destination message queue or that identifies a catalog record from which the IPM Manager can obtain the destination queue address. Set this field to `nil` if you intend to use the `IPMAddRecipient` function to add all the recipient addresses later.
`replyQueue`	A pointer to an `OCERecipient` structure that specifies the queue in which you receive your incoming messages. The `OCERecipient` structure can specify the reply queue directly, or can specify a record in a catalog that contains the reply queue information. If you specify `nil` for this field and a local identity for the `identity` field, the IPM Manager uses the PowerTalk Setup catalog to fill in the reply queue field in the message header at the time the message is sent. You can also set this field to `nil` if you intend to use the `IPMAddReplyQueue` function to specify the reply queue later.
`procHint`	A pointer to a process hint, which is a string of up to 32 characters, reserved for your use. The IPM Manager puts this string into the message header. You can use this field, for example, to provide information that helps your recipients determine how to process the message.
`msgType`	A pointer to an `IPMMsgType` structure, which specifies the type of message that you are creating. The IPM Manager and other AOCE components do not read the message type; it is for the use of applications only. Note, however, that the Finder might display the contents of the message header's message-type field if the user displays the Info dialog box for the message while the message is in the Out Tray.

`refCon`	An unsigned 4-byte number, reserved for your use. The IPM Manager puts this value into the message header. You can use this field, for example, to indicate that the message has content of some particular type.
`newMsgRef`	A reference number returned by the function. You must use this number when you call other functions to complete the process of creating the message.
`identity`	The authentication identity of the creator of the message. If you provide a nonzero value for this field, the IPM Manager uses this information to fill in the `sender` field in the message header.
`sender`	A pointer to an `IPMSender` structure, which identifies the sender of the message. If you specify 0 or a local identity for the `identity` field, you should provide a meaningful value for the sender. For an application-to-application message, you might use the application name as the sender. To specify an individual as a sender, you can put the user's name in the `IPMSender` structure or you can provide the record ID of the sender's user record. If you specify a specific identity for the `identity` field, the function ignores the `sender` field.

DESCRIPTION

You must call the `IPMNewMsg` function to begin the process of creating a new message that is to be sent to a recipient. (Use the `IPMNewHFSMsg` function to start a message to be saved on disk.) The IPM Manager uses information that you provide in the parameter block of `IPMNewMsg` to fill in fields of the message header of the new message.

The IPM Manager uses the information you provide in the `recipient` field to determine where to send the message and returns any delivery or nondelivery reports to the queue that you specify in the `replyQueue` field. If you do not know the recipient at the time you call `IPMNewMsg` function, or if you have more than one recipient, you can use the `IPMAddRecipient` function to provide the recipients. If you do not know the reply queue at the time you call the `IPMNewMsg` function, you can use the `IPMAddReplyQueue` function to add the reply queue later.

If the `recipient` or `replyQueue` fields specify a record in a PowerShare catalog, the IPM Manager looks up the catalog records at the time it sends the message.

Note
Because the PowerShare server acts as a trusted agent when resolving addresses in catalogs, the sender of the message need not have the access privileges necessary to read these addresses. ◆

The IPM Manager uses any specific identity you provide in the `identity` field to fill in the `sender` field in the message header. If the IPM Manager and each intervening store-and-forward server can authenticate the message, the recipient can then rely on the `sender` field to indicate the authenticated originator of the message. If you specify 0 or a local identity for the `identity` field, then you should provide a meaningful value for the `sender` field, such as the name of the originator of the message.

Note
If you specify either a local identity or 0 for the `identity` field, the IPM Manager stores the message on the local computer until transmitting it. If you provide a specific identity, the IPM Manager creates the message on the computer containing the PowerShare server to which that identity provides access. ◆

You can use the `SDPPromptForIdentity` function to obtain an identity for the originator of the message. This function allows the user to decide whether to provide a local identity, a specific identity, or no identity (guest access). The `SDPPromptForIdentity` function returns to your application the identity plus a value that tells you which kind of identity it is. To obtain a local identity without displaying a dialog box, use the `AuthGetLocalIdentity` function.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0402

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidMsgType`	–15091	Message type is invalid
`kIPMInvalidProcHint`	–15092	Process hint is invalid
`kIPMMsgTypeReserved`	–15095	Message type reserved for system use
`kIPMNestedMsgOpened`	–15097	Nested message opened; cannot do operation
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMCorruptDataStructures`	–15099	Message is corrupt
`kIPMInvalidSender`	–15103	Sender is invalid
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

See “Message Addressing Structures” on page 7-24 for more detailed information about the format and contents of an `OCERecipient` structure.

The `IPMSender` structure is described on page 7-39.

You can use the `IPMAddRecipient` function (page 7-50) to add recipient addresses to a message.

You can use the `IPMAddReplyQueue` function (page 7-52) to specify the reply queue.

See “Message and Block Types” on page 7-26 for more information about the `IPMMsgType` structure.

The `RString` structure and record IDs are described in the chapter “Introduction to the Apple Open Collaboration Environment” in this book.

You can use the `SDPPromptForIdentity` function to obtain an identity. That function is described in the chapter “Standard Catalog Package” in this book. You can use the `AuthGetLocalIdentity` function to obtain a local identity. See the chapter “Authentication Manager” in this book for a description of the `AuthGetLocalIdentity` function.

Use the `IPMNewHFSMsg` function, described next, to start a message to be saved on disk.

IPMNewHFSMsg

The `IPMNewHFSMsg` function starts the process of creating a new message to be saved as an HFS file on disk.

```
pascal OSErr IPMNewHFSMsg(IPMParamBlockPtr paramBlock,
                          Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`hfsPath`	`FSSpec*`	Specifier of the file in which to save the message.
→	`recipient`	`OCERecipient*`	Pointer to the recipient’s queue address.
→	`replyQueue`	`OCERecipient*`	Pointer to the queue address for message replies.
→	`procHint`	`StringPtr`	Pointer to a character string for your use.
→	`msgType`	`IPMMsgType*`	Pointer to the message type.
→	`refCon`	`unsigned long`	Reserved for your use.
←	`newMsgRef`	`IPMMsgRef`	Message reference number.
→	`identity`	`AuthIdentity`	Authentication identity.
→	`sender`	`IPMSender*`	Pointer to the sender’s name.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`hfsPath`	A pointer to the file system specification structure that describes the file in which you wish to save the message.
`recipient`	A pointer to an `OCERecipient` structure that either specifies a destination message queue or that identifies a catalog record from which the IPM Manager can obtain the destination queue address. Set this field to `nil` if you intend to use the `IPMAddRecipient` function to add all the recipient addresses later.
`replyQueue`	A pointer to an `OCERecipient` structure that specifies the queue in which you receive your incoming messages. The `OCERecipient` structure can specify the reply queue directly, or can specify a record in a catalog that contains the reply queue information. Set this field to `nil` if you intend to use the `IPMAddReplyQueue` function to specify the reply queue later.
`procHint`	A pointer to a process hint, which is a string of up to 32 characters, reserved for your use. The IPM Manager puts this string into the message header. You can use this field, for example, to provide information that helps your recipients determine how to process the message.
`msgType`	A pointer to an `IPMMsgType` structure, which specifies the type of message that you are creating. The IPM Manager and other AOCE components do not read the message type; it is for the use of applications only.
`refCon`	An unsigned 4-byte number, reserved for your use. The IPM Manager puts this value into the message header. You can use this field, for example, to indicate that the message has content of some particular type.
`newMsgRef`	A reference number returned by the function. You must use this number when you call other functions to complete the process of creating the message.
`identity`	The authentication identity of the creator of the message. If you provide a nonzero value for this field, the IPM Manager uses this information to fill in the `sender` field in the message header.
`sender`	A pointer to an `IPMSender` structure, which identifies the sender of the message. If you specify 0 or a local identity for the `identity` field, you should provide a meaningful value for the sender. For an application-to-application message, you might use the application name as the sender. To specify an individual as a sender, you can put the user's name in the `IPMSender` structure or you can provide the record ID of the sender's user record. If you specify a specific identity value for the `identity` field, the function ignores the `sender` field.

DESCRIPTION

You must call the `IPMNewHFSMsg` function to begin the process of creating a new message that is to be saved to a file on disk. (Use the `IPMNewMsg` function to start a message to be sent to a recipient.) The IPM Manager fills in fields of the message header of the new message from information that you provide in the parameter block of the `IPMNewHFSMsg` function.

The IPM Manager uses any specific identity you provide in the `identity` field to fill in the `sender` field in the message header. If you specify 0 or a local identity for the `identity` field, then you should provide a meaningful value for the `sender` field, such as the name of the originator of the message.

Note
The IPM Manager does not provide any way to send a message that has been saved on disk. If you want to send a message and in addition save it to disk, you must build the message twice, once using the `IPMNewHFSMsg` function and once using the `IPMNewMsg` function. ◆

You can use the `SDPPromptForIdentity` function to obtain an identity for the originator of the message. This function allows the user to decide whether to provide a local identity, a specific identity, or no identity (guest access). The `SDPPromptForIdentity` function returns to your application the identity plus a value that tells you which kind of identity it is. To obtain a local identity without displaying a dialog box, use the `AuthGetLocalIdentity` function.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$041E

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter

SEE ALSO

See “Message Addressing Structures” on page 7-24 for more detailed information about the format and contents of an `OCERecipient` structure.

The `IPMSender` structure is described on page 7-39.

You can use the `IPMAddRecipient` function (page 7-50) to add recipient addresses to a message.

You can use the `IPMAddReplyQueue` function (page 7-52) to specify the reply queue.

See "Message and Block Types" on page 7-26 for more information about the `IPMMsgType` structure.

The `RString` structure and record IDs are described in the chapter "Introduction to the Apple Open Collaboration Environment" in this book.

You can use the `SDPPromptForIdentity` function to obtain an identity. That function is described in the chapter "Standard Catalog Package" in this book. You can use the `AuthGetLocalIdentity` function to obtain a local identity. See the chapter "Authentication Manager" in this book for a description of the `AuthGetLocalIdentity` function.

Use the `IPMNewMsg` function (page 7-43) to start a message to be sent.

IPMAddRecipient

The `IPMAddRecipient` function adds a recipient to a new message that you are creating.

```
pascal OSErr IPMAddRecipient(IPMParamBlockPtr paramBlock,
                              Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`recipient`	`OCERecipient*`	Pointer to the recipient's queue address.

Field descriptions

`msgRef`	The message reference number of the message or nested-message block to which you want to add a recipient. This number is returned by the `IPMNewMsg` function for a message you intend to send, by the `IPMNewHFSMsg` function for a message you intend to save to disk, and by the `IPMNewNestedMsgBlock` function for a nested-message block.

`recipient`	A pointer to an `OCERecipient` structure that either specifies a destination message queue or that identifies a catalog record from which the IPM Manager can obtain the destination queue address.

DESCRIPTION

You can call the `IPMAddRecipient` function at any time during the message-creation process to add a recipient to the message. You repeat this call for each recipient that you add to the message (except for recipients that belong to a group; see "Message Addressing Structures" on page 7-24). You can add only one recipient to a message when you call the `IPMNewMsg` or `IPMNewHFSMsg` function; if you want to add more than one recipient to a message, you must call the `IPMAddRecipient` function.

When you call the `IPMAddRecipient` function for a new message, the function adds the specified recipient to the message header. If you are working with a nested message, the function adds the recipient to the header of the nested message.

If the `recipient` parameter specifies a record in a catalog, the `IPMAddRecipient` function does not look up the address of the recipient in the catalog. The IPM Manager looks up catalog records when you send the message.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_oceTBDispatch`	$0403

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMCorruptDataStructures`	–15099	Message is corrupt

SEE ALSO

See "Message Addressing Structures" on page 7-24 for more detailed information about the format and contents of an `OCERecipient` structure.

IPMAddReplyQueue

The `IPMAddReplyQueue` function adds the reply queue to the header of a new message that you are creating.

```
pascal OSErr IPMAddReplyQueue(IPMParamBlockPtr paramBlock,
                              Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async` A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`replyQueue`	`OCERecipient*`	Pointer to the queue address for message replies.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef` The message reference number of the message or nested-message block to which you want to add the reply queue. This number is returned by the `IPMNewMsg` function for a message you intend to send, by the `IPMNewHFSMsg` function for a message you intend to save to disk, and by the `IPMNewNestedMsgBlock` function for a nested-message block.

`replyQueue` A pointer to an `OCERecipient` structure that specifies the queue in which you receive your incoming messages. The `OCERecipient` structure can specify the reply queue directly or specify a record in a catalog that contains the reply queue information.

If you specify `nil` for this field and specified a local identity for the `identity` field in the `IPMNewMsg` function, the IPM Manager uses the PowerTalk Setup catalog to fill in the reply queue field in the message header at the time the message is sent.

DESCRIPTION

You can call the `IPMAddReplyQueue` function at any time during the message-creation process. When you call the `IPMAddRecipient` function for a new message, the function adds the specified reply queue to the message header. If you are working with a nested message, the function adds the reply queue to the header of the nested message.

Each message or nested message has only one reply queue. If you have already specified a reply queue for the message that you specify in the `msgRef` field, the `IPMAddReplyQueue` function returns the `kOCEParamErr` result code.

If the `replyQueue` parameter specifies a record in a catalog, the `IPMAddRecipient` function does not look up the address of the reply queue in the catalog. The IPM Manager resolves addresses in catalog records at the time a message is sent.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$041D

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMCorruptDataStructures`	–15099	Message is corrupt

SEE ALSO

See “Message Addressing Structures” on page 7-24 for more detailed information about the format and contents of an `OCERecipient` structure.

IPMNewBlock

The `IPMNewBlock` function creates a new block at the end of the message or nested message that you are currently recording and returns the offset to its starting point.

```
pascal OSErr IPMNewBlock(IPMParamBlockPtr paramBlock,
                         Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async` A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`blockType`	`IPMBlockType`	Type of block you are adding.
→	`refCon`	`unsigned long`	Reserved for your use.
←	`startingOffset`	`long`	Offset to new block.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message or nested message to which you want to add the block. This number is returned by the `IPMNewMsg` function for a message you intend to send, by the `IPMNewHFSMsg` function for a message you intend to save to disk, and by the `IPMNewNestedMsgBlock` function for a nested message.
`blockType`	A pointer to an `IPMBlockType` data type that specifies the type of block that you are adding to the message.
`refCon`	An unsigned 4-byte number, reserved for your use. The IPM Manager puts this value into the TOC field of the message header. You can use this field, for example, to identify block subtypes for your own use.
`startingOffset`	The offset, in bytes, from the start of the message body to the start of the new block. This value is returned by the function. You can use this offset as a starting point when you call the `IPMWriteMsg` function to add data to the block.

DESCRIPTION

You can call the `IPMNewBlock` function at any time during the message-creation process to create a new message block.

The `IPMNewBlock` function creates the new block at the end of the message, records the offset to the new block, and then returns the offset to you. You can use this value to determine the offset to provide to the `IPMWriteMsg` function when you add data to the block or overwrite data in the block.

Note

The IPM Manager does not allow you to modify the starting point of a block. When you call the `IPMNewBlock` function to create a new block, you freeze the size of the previous block. You can use the `IPMWriteMsg` function to overwrite data in an existing block, but if you try to write more data than was originally in the block, you write over the block boundary into the following block. ◆

A nested message is contained entirely within a single block of the enclosing message and has exactly the same structure as any other message (see Figure 7-2 on page 7-5). Once you have called the `IPMNewNestedMsgBlock` function to start a nested message, you must call the `IPMEndMsg` function to end the nested message before adding another block to the outer message. If you specify the message reference of an outer message before completing a nested message, the `IPMNewBlock` function returns the `kIPMNestedMsgOpened` result code.

SPECIAL CONSIDERATIONS

If you specify `kIPMSignature` as the creator of the block in the `IPMBlockType` data type, the function returns the `kIPMMsgTypeReserved` result code.

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0404

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMMsgTypeReserved`	–15095	The `blockType` parameter specifies a block type reserved for system use
`kIPMNestedMsgOpened`	–15097	The message reference in the `msgRef` parameter specifies an outer message, but nested message is not yet closed
`kIPMCorruptDataStructures`	–15099	Message is corrupt

SEE ALSO

The `IPMBlockType` data type is defined on page 7-28.

You start a new message by calling the `IPMNewMsg` function (page 7-43) or the `IPMNewHFSMsg` function (page 7-47). You start a new nested message by calling the `IPMNewNestedMsgBlock` function (next).

You can use the `IPMWriteMsg` function (page 7-61) to add data to the block.

IPMNewNestedMsgBlock

The `IPMNewNestedMsgBlock` function starts a new nested message from information that you provide to the function. Use this function to begin recording a new nested message that you create from scratch.

```
pascal OSErr IPMNewNestedMsgBlock(IPMParamBlockPtr paramBlock,
                                  Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`recipient`	`OCERecipient*`	Pointer to the recipient’s queue address.
→	`replyQueue`	`OCERecipient*`	Pointer to the queue address for message replies.
→	`procHint`	`StringPtr`	Pointer to character string for your use.
→	`msgType`	`IPMMsgType*`	Pointer to the message type.
→	`refCon`	`unsigned long`	Reserved for your use.
←	`newMsgRef`	`IPMMsgRef`	Message reference number.
←	`startingOffset`	`long`	Offset to the start of the nested message.
→	`identity`	`AuthIdentity`	Authentication identity.
→	`sender`	`IPMSender*`	Pointer to sender’s name.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message or nested message into which you want to insert the new nested message. This number is returned by the `IPMNewMsg` function for a message you intend to send, by the `IPMNewHFSMsg` function for a message you intend to save to disk, and by the `IPMNewNestedMsgBlock` function for a nested message.
`recipient`	A pointer to an `OCERecipient` structure that either specifies a destination message queue or that identifies a catalog record from which the IPM Manager can obtain the destination queue address. Set this field to `nil` if you intend to use the `IPMAddRecipient` function to add all the recipient addresses later.

`replyQueue`	A pointer to an `OCERecipient` structure that specifies the queue in which you receive your incoming message reports. In most cases, you have only one message queue. Set this field to `nil` if you intend to use the `IPMAddReplyQueue` function to specify the reply queue later.
`procHint`	A pointer to a process hint, which is a string of up to 32 characters, reserved for your use. The IPM Manager puts this string into the nested-message header. You can use this field, for example, to provide information that helps your recipients determine how to process the nested message.
`msgType`	A pointer to an `IPMMsgType` structure, which specifies the type of nested message that you are creating. This value is application dependent and is not read by any AOCE component.
`refCon`	An unsigned 4-byte number, reserved for your use. The IPM Manager puts this value into the nested-message header.
`newMsgRef`	A reference number returned by the function. You must use this number when you call the `IPMAddRecipient`, `IPMAddReplyQueue`, `IPMNewBlock`, `IPMWriteMsg`, `IPMNestMsg`, `IPMNewNestedMsgBlock`, and `IPMEndMsg` functions to complete the process of creating this nested message.
`startingOffset`	The offset in bytes to the start of the new nested-message block from the start of the enclosing message body. This value is returned by the function.
`identity`	The authentication identity of the creator of the message. If you provide a nonzero value for this field, the IPM Manager uses this information to fill in the `sender` field in the message header.
`sender`	A pointer to an `IPMSender` structure, which identifies the sender of the message. If you specify 0 or a local identity for the `identity` field, you should provide a meaningful value for the sender. For an application-to-application message, you might use the application name as the sender. To specify an individual as a sender, you can put the user's name in the `IPMSender` structure or you can provide the record ID of the sender's user record. If you specify a specific identity in the `identity` field, the function ignores the `sender` field.

DESCRIPTION

You can call the `IPMNewNestedMsgBlock` function at any time during the message-creation process to start a new nested message.

The `IPMNewNestedMsgBlock` function first creates a new block at the end of the message. The `msgCreator` field of the block type of the new block is equal to the constant `kIPMSignature` and the `msgType` field is equal to `kIPMEnclosedMsgType`. The `IPMNewNestedMsgBlock` function then fills in fields of the message header of the new nested message from information that you provide in the parameter block of the `IPMNewNestedMsgBlock` function.

Note that, because the header of the nested message is located within a block of the enclosing message, the IPM Manager does not read the nested-message header and so does not use the information in its message-delivery process.

The IPM Manager uses any specific identity you provide in the `identity` field to fill in the `sender` field in the message header. If you specify 0 or a local identity for the `identity` field, then you should provide a meaningful value for the `sender` field, such as the name of the originator of the message.

After you call the `IPMNewNestedMsgBlock` function to start a nested message, you can call the `IPMNewBlock` function to add a new block to the nested message, the `IPMNewNestedMsgBlock` function to nest another message within the nested message, or any of the functions that add information to the message header or to the body of the message. When you call any of these functions, you must pass the message reference value returned by the `IPMNewNestedMsgBlock` function.

You must call the `IPMEndMsg` function to complete the nested message before you can add any more information to the enclosing message. After you call the `IPMEndMsg` function to end the nested message, you cannot add any recipients or blocks to the nested message.

SPECIAL CONSIDERATIONS

Although the IPM Manager allows you to add any number of nested-message blocks at the same nesting level in a message, the MSAM interface does not support this feature. Therefore, if you want your message to be compatible with MSAMs, you must not add more than one nested-message block at a given level of nesting. You can, however, nest a message within another nested message to as many nesting levels as disk and memory resources allow.

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0405

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidMsgType`	–15091	Message type is invalid
`kIPMInvalidProcHint`	–15092	Process hint is invalid
`kIPMCorruptDataStructures`	–15099	Message is corrupt
`kIPMInvalidSender`	–15103	Sender is invalid

SEE ALSO

See “Message Addressing Structures” on page 7-24 for more detailed information about the format and contents of an `OCERecipient` structure.

The `IPMSender` structure is described in “Sender Structure” on page 7-39.

You can use the `IPMAddRecipient` function (page 7-50) to add recipient addresses to a message.

You can use the `IPMAddReplyQueue` function (page 7-52) to specify the reply queue.

See “Message and Block Types” on page 7-26 for more information about the `IPMMsgType` structure.

IPMNestMsg

The `IPMNestMsg` function creates a new block at the end of the specified new message and stores the existing message that you specify into the new block.

```
pascal OSErr IPMNestMsg(IPMParamBlockPtr paramBlock,
                        Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`refCon`	`unsigned long`	Reserved for your use.
→	`msgToNest`	`IPMMsgRef`	Message reference number of the message to nest.
←	`startingOffset`	`long`	Offset to the start of the nested message.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message to which you want to add a nested message. This number is returned by the `IPMNewMsg` function for a message you intend to send, by the `IPMNewHFSMsg` function for a message you intend to save to disk, and by the `IPMNewNestedMsgBlock` function for a nested message.

`refCon`	An unsigned 4-byte number, reserved for your use. The IPM Manager puts this value into the nested-message header.
`msgToNest`	This parameter contains the message reference number of an existing message that you want to nest within the message that you specify in the `msgRef` field. This number is returned by the `IPMOpenMsg` function for a message you have read, by the `IPMOpenHFSMsg` function for a message you have read from disk, or by the `IPMOpenBlockAsMsg` for a nested message.
`startingOffset`	The offset, in bytes, to the start of the new nested-message block from the start of the body of the enclosing message. You can use this value if you want to create your own table of contents for a message you are creating.

DESCRIPTION

You can call the `IPMNestMsg` function at any time during the message-creation process. The `IPMNestMsg` function adds an existing message as a nested message at the end of the message that you specify in the `msgRef` field. Before you call the `IPMNestMsg` function, you must use the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function to open the message to be nested.

The `IPMNestMsg` function first creates a new block at the end of the message. The `msgCreator` field of the block type of the new block is equal to the constant `kIPMSignature` and the `msgType` field is equal to `kIPMEnclMsgType`. The function then writes the specified message into the new block.

The `IPMNestMsg` function returns, in the `startingOffset` parameter, the offset to the start of the new block. The function provides this offset for your information only. You should not call `IPMWriteMsg` to make changes to this nested message.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_oceTBDispatch`	$0406

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidMsgType`	–15091	Message type is invalid
`kIPMInvalidProcHint`	–15092	Process hint is invalid
`kIPMMsgTypeReserved`	–15095	Message type reserved for system use
`kIPMNestedMsgOpened`	–15097	Nested message opened; cannot do operation
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMCorruptDataStructures`	–15099	Message is corrupt
`kIPMInvalidSender`	–15103	Sender is invalid
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

See “Message and Block Types” on page 7-26 for more information about the `IPMMsgType` structure.

You can obtain a message reference number from the `IPMOpenMsg` function (page 7-82), the `IPMOpenHFSMsg` function (page 7-84), or the `IPMOpenBlockAsMsg` (page 7-86).

IPMWriteMsg

The `IPMWriteMsg` function writes data to the specified location within the body of a message.

```
pascal OSErr IPMWriteMsg(IPMParamBlockPtr paramBlock,
                         Boolean async);
```

`paramBlock`
: A pointer to a parameter block.

`async`
: A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`mode`	`IPMAccessMode`	The mode in which the function interprets the offset value.
→	`offset`	`long`	Offset at which to begin writing.
→	`count`	`unsigned long`	Number of bytes of data to write.
→	`buffer`	`Ptr`	Pointer to the data buffer.
←	`actualCount`	`unsigned long`	Number of bytes of data written.
→	`currentBlock`	`Boolean`	Set to `true` to restrict writing to the current block.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message or nested message to which you want to write. This number is returned by the `IPMNewMsg` function for a message you intend to send, by the `IPMNewHFSMsg` function for a message you intend to save to disk, and by the `IPMNewNestedMsgBlock` function for a nested message.
`mode`	The mode in which the `offset` parameter is to be interpreted. The function uses this field to determine whether to begin writing data at the end of the last data written or to use the offset value to calculate another starting point relative to the beginning of the message, the end of the message, or the current location. See the discussion following these field descriptions for details.
`offset`	An offset that the function uses when it calculates the starting point of the write operation. See the following discussion for details.
`count`	The number of bytes of data that you want the function to write from the buffer into the message.
`buffer`	A pointer to your data buffer.
`actualCount`	The number of bytes of data the function actually wrote into the message.
`currentBlock`	A Boolean value that specifies whether you want the entire write operation to occur within the current block. The ***current block*** is always the last block to be added to the message. If you set this field to `true` but the values you specify for the `mode` and `offset` fields require the function to write data into another block, the function cancels the write operation and returns the `kIPMInvalidOffset` result code.

DESCRIPTION

The IPM Manager uses a marker (referred to as the ***message mark***) that points to the current location within a message that you are creating. After the `IPMNewBlock` function completes, the message mark points to the first byte in the new block. After the `IPMWriteMsg` function completes, the mark points to the end of the last byte written.

Note

The way you use the message mark, mode, and offset to read and write messages is similar to the way you use the file mark, positioning mode, and positioning offset to read and write files. See *Inside Macintosh: Files* for more information about how the File Manager treats these parameters. ◆

You use the `mode` and `offset` parameters to specify the point in the message at which the `IPMWriteMsg` function starts writing. The `mode` parameter indicates whether you want the `IPMWriteMsg` function to begin writing at the current position of the mark or to calculate another starting point relative to the beginning of the message, the end of the message, or the current mark location. (In the case of a nested message, offsets are relative to the start or end of the nested message, not the enclosing message.) You can set the `mode` parameter to any one of the following values:

```
enum {
   kIPMAtMark,
   kIPMFromStart,
   kIPMFromLEOM,
   kIPMFromMark
};
```

Constant descriptions

`kIPMAtMark`	The `IPMWriteMsg` function starts writing at the current position of the mark. In this case, the function ignores the offset value. This mode is useful, for example, for writing data in sequence into a new block.
`kIPMFromStart`	If the `currentBlock` parameter is set to `true`, the function interprets the value in the `offset` parameter as an offset from the beginning of the current block. If the `currentBlock` parameter is set to `false`, the function interprets the value in the `offset` parameter as an offset from the beginning of the message body. If you want to start writing at the beginning of the second block in the message, for example, you can set `currentBlock` to `false` and use the offset that the `IPMNewBlock` function returned when you created the second block. When you use this mode, you cannot set the `offset` parameter to a negative value.
`kIPMFromLEOM`	The function interprets the value in the `offset` parameter as an offset from the current end of the message.

`kIPMFromMark`	The function interprets the value in the `offset` parameter as an offset from the current position of the mark. Use a negative offset value to indicate a starting point prior to the current position of the mark and a positive offset value to indicate a starting point following the current position of the mark.

If the mark is at the end of the last block, the function extends the end of the block and the end of the message as it writes data into the block.

Note
If you use a positive offset to position the mark past the current end of the message, the function extends the end of the message and writes the data in the location you requested. In this case, you incorporate into the message whatever happened to be on disk between the previous end of the message and the location at which you start writing. ◆

If you set the `currentBlock` parameter to `true`, the `IPMWriteMsg` function returns an error rather than starting to write in a block other than the last block to be added to the message.

Note that the IPM Manager places the offset to each block in the message header when you first create the block. You cannot change this information in the message header after the block is created. Therefore, when you call the `IPMNewBlock` function to create a new block, you freeze the size of the previous block. You can use the `IPMWriteMsg` function to write over data in an existing block, but you cannot change the size of the block. If you write too much data to fit in an existing block, the function writes over the block boundary into the following block.

When you call the `IPMWriteMsg` function, it first calculates the starting position of the write request. The function then checks the value of the `currentBlock` parameter to determine if it is in conflict with the starting position. That is, if you set `currentBlock` to `true` and specify a write location that falls in another block of the message, the `IPMWriteMsg` function returns the `kIPMInvalidOffset` error.

If the `currentBlock` setting is not in conflict with the specified starting position, the function writes the data from the buffer into the message and returns, in the `actualCount` field, the number of bytes written.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0407

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMNotInABlock`	–15096	The specified starting point does not fall within the body of the message
`kIPMNestedMsgOpened`	–15097	The message reference in the `msgRef` parameter specifies an outer message, but nested message is not yet closed
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

The `IPMNewMsg` function (page 7-43), the `IPMNewHFSMsg` function (page 7-47), and the `IPMNewNestedMsgBlock` function (page 7-56) all return message reference numbers.

The `IPMNewBlock` function (page 7-53) and the `IPMNewNestedMsgBlock` function (page 7-56) return the offset to the start of a new block.

IPMEndMsg

The `IPMEndMsg` function ends the message-creation process for the message or nested message that you specify. It can also provide a digital signature for the message.

```
pascal OSErr IPMEndMsg(IPMParamBlockPtr paramBlock,
                       Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async`
A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
←	`msgID`	`IPMMsgID`	Message ID.
→	`msgTitle`	`RString*`	Message title.
→	`deliveryNotification`	`IPMNotificationType`	Delivery report specifier.
→	`priority`	`IPMPriority`	Message priority.
→	`cancel`	`Boolean`	Cancel the message?
→	`signature`	`SIGSignaturePtr`	Pointer to a digital signature.
→	`signatureSize`	`Size`	Size of the digital signature.
→	`signatureContext`	`SIGContextPtr`	Pointer to digital signature context.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message or nested message that you want to complete. This number is returned by the `IPMNewMsg` function for a message you intend to send, by the `IPMNewHFSMsg` function for a message you intend to save to disk, and by the `IPMNewNestedMsgBlock` function for a nested message.
`msgID`	The message ID, a unique identifier assigned to the message by the IPM Manager. You can use this value to identify a message.
`msgTitle`	The message title. Because the Finder displays this title for the user for any message in the Out Tray, the message title should reflect the subject, contents, or purpose of the message. The maximum size of this title is 32 bytes (that is, an `RString32` structure).
`deliveryNotification`	The types of delivery reports you want to receive. See "Delivery Notification," beginning on page 7-28, for more information about this value.
`priority`	The priority of the message. Set this parameter to any one of the following values: `kIPMNormalPriority`, `kIPMLowPriority`, or `kIPMHighPriority`.
`cancel`	A Boolean value that specifies whether to cancel the message. Set this field to `true` to cancel the message or to `false` to send the message. If the `IPMEndMsg` function applies to a nested message, the function ignores the value of this field.
`signature`	A pointer to a digital signature. You must allocate a buffer for the signature and pass a pointer to it in this field. If you specify `nil` for the `signatureContext` field, the function ignores the `signature` field. See the following discussion for more information about digital signatures.
`signatureSize`	The size of the digital signature. This value is returned by the `SIGSignPrepare` function.
`signatureContext`	A pointer to the signature context you obtained from the `SIGNewContext` function and provided to the `SIGSignPrepare` function. Specify `nil` for this pointer if you do not want a digital signature added to the message.

DESCRIPTION

When you call the `IPMEndMsg` function, it checks the setting of the `cancel` parameter to see if you are canceling the message. If so, the function destroys the message. Otherwise, the function completes the message-creation process for the specified message. If the message reference number you specify applies to a nested-message block, the IPM Manager ends the nested-message block and applies any subsequent functions that you call to the enclosing message. The enclosing message can be another nested message or the top-level message (that is, the message you started with the `IPMNewMsg` or `IPMNewHFSMsg` function). To completely finish the message-creation process, you must call the `IPMEndMsg` function once for each nested message and once for the top-level message.

IMPORTANT
You cannot close an enclosing message until any messages nested within it have been closed. ▲

Once you have called the `IPMEndMsg` function to close the top-level message, you cannot make any more changes to the message. If you created the message with the `IPMNewHFSMsg` function, the IPM Manager saves the message to the disk file you specified when you called the `IPMNewHFSMsg` function. If you created the message with the `IPMNewMsg` function, the IPM Manager sends the message to each recipient and generates any requested reports.

The IPM Manager uses the value of the `deliveryNotification` parameter to determine when to generate report messages and whether to include the original message in any reply messages that are returned by the recipients.

If you want to add a digital signature to the message, you must call the `SIGNewContext` and `SIGSignPrepare` functions before you call the `IPMEndMsg` function. You can then allocate a buffer for the signature, or specify `nil` for the `signature` parameter, in which case the Digital Signature Manager allocates the buffer for you on your application heap. The size needed for the buffer is returned by the `SIGSignPrepare` function. Pass a pointer to the buffer in the `signature` parameter to the `IPMEndMsg` function, the size of the buffer in the `signatureSize` parameter, and a pointer to the signature context (returned by the `SIGNewContext` function) in the `signatureContext` parameter.

Note
If you are adding a digital signature to a large message, the `IPMEndMsg` function can take a long time to complete (up to several minutes on some computers). You should display a dialog box informing the user of this possibility. ◆

The `IPMEndMsg` function places the signature in a block with a creator of `kIPMSignature` and a type of `kIPMDigitalSignature`. A message can contain only one block of this type, and you must use the `IPMEndMsg` function to create this block.

Note
The signature context used to create a digital signature has no relationship to the contexts discussed in "Managing Message Queues" starting on page 7-68 and elsewhere in this chapter. ◆

SPECIAL CONSIDERATIONS

If you want to add a digital signature to the message (that is, you pass a non-`nil` value for the `signatureContext` parameter), you must call the `IPMEndMsg` function synchronously. There must also be at least 8.5 KB of stack space available.

If you pass `nil` for the `signatureContext` parameter, there must be enough space in your application heap to hold the signature.

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0408

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidOffset`	–15093	Bad offset for read or write operation
`kIPMNestedMsgOpened`	–15097	The message reference in the `msgRef` parameter specifies an outer message, but nested message is not yet closed
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMCorruptDataStructures`	–15099	Message is corrupt
`kIPMAbortOfNestedMsg`	–15100	Adding nested message was canceled
`kIPMInvalidSender`	–15103	Sender is invalid
`kIPMNoRecipientsYet`	–15104	Require recipient to send
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

You start creating a message with the `IPMNewMsg` function (page 7-43) or the `IPMNewHFSMsg` function (page 7-47), and start a nested-message block with the `IPMNewNestedMsgBlock` function (page 7-56).

See "Delivery Notification," beginning on page 7-28, for more information about the delivery notification flag byte.

Digital signatures and the `SIGNewContext` and `SIGSignPrepare` functions are discussed in the chapter "Digital Signature Manager" in this book.

Managing Message Queues

You can create any number of local input message queues for your own use. This section describes the functions you can use to create input message queues, open queues, enumerate their contents, and close them.

IPMCreateQueue

The `IPMCreateQueue` function creates a physical queue at the specified location.

```
pascal OSErr IPMCreateQueue(IPMParamBlockPtr paramBlock,
                            Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`queue`	`OCERecipient*`	Name and location of the new queue.
→	`identity`	`AuthIdentity`	Authentication identity.
→	`owner`	`PackedRecordID*`	Owner of the queue.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`queue`	A pointer to an `OCERecipient` structure that specifies the name and location of the new queue. You must use the queue-name form of the `OCERecipient` structure for this field.
`identity`	The authentication identity of the creator of the queue. If you are creating the queue on a server computer, the messaging server uses this identity to verify that you have the privileges necessary to create a queue. Only the administrator of that server can create queues. The function ignores this field if you specify the local computer as the location of the new queue.
`owner`	A pointer to the packed record ID of the owner of the queue. If you are creating a queue on a remote computer, you must specify an owner of the queue in this field. Only the creator of the queue and the owner of the queue can open or delete the queue. The function ignores this field if you specify the local computer as the location of the new queue.

DESCRIPTION

You can create a new queue at any time. You can create a queue on the local computer or on a server computer.

IMPORTANT
You should use restraint in creating queues because the IPM Manager provides no interface for listing and managing queues. Also, each queue uses memory and disk resources. ▲

Once you have used the `IPMCreateQueue` function to create a physical queue, you must open one or more virtual queues to list and open the messages in the queue. Use the `IPMOpenQueue` function to open a virtual queue.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0411

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMBadQName`	–15112	Invalid queue name

SEE ALSO

"Message Addressing Structures" on page 7-24 defines the `OCERecipient` structure. The queue-name form of this structure is described in "Queue-Name Format for Attribute Values" on page 7-16.

You must use the `IPMOpenQueue` function (page 7-72) to open a queue before you can open the messages in the queue. You must have an open queue context before you can open a queue; use the `IPMOpenContext` function, described next, to open a context.

IPMOpenContext

The `IPMOpenContext` function creates a new queue context.

```
pascal OSErr IPMOpenContext(IPMParamBlockPtr paramBlock,
                            Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
←	`contextRef`	`IPMContextRef`	Context reference number.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`contextRef`	The context reference number for the new context. You must use this number when opening a queue or closing the context.

DESCRIPTION

You must specify a context reference number when you open a virtual queue. You must specify a virtual-queue reference number when you open a message. When you close a context, the IPM Manager closes all of the virtual queues that belong to that context and all of the open messages that belong to those queues. You can create as many contexts as you wish; in any case, you must call the `IPMOpenContext` function at least once to obtain a context reference number before you can open any queues.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0400

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter

SEE ALSO

Use the `IPMOpenQueue` function, discussed next, to open a virtual queue and add it to a context.

Use the `IPMCloseContext` function (page 7-77) to close all the virtual queues and open messages associated with a context.

IPMOpenQueue

The `IPMOpenQueue` function opens the specified queue and associates it with the specified context.

```
pascal OSErr IPMOpenQueue(IPMParamBlockPtr paramBlock,
                           Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async` A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`contextRef`	`IPMContextRef`	Context reference number.
→	`queue`	`OCERecipient*`	Queue that you want to open.
→	`identity`	`AuthIdentity`	Authentication identity.
→	`filter`	`IPMFilter*`	Pointer to the queue filter.
←	`newQueueRef`	`IPMQueueRef`	Virtual-queue reference number.
→	`notificationProc`	`IPMNoteProcPtr`	Reserved; set to `nil`.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`contextRef` A context reference number. When you close the context specified by this reference number, the IPM Manager closes all of the virtual queues that you opened using this reference number.

`queue` A pointer to an `OCERecipient` structure that specifies the name and location of the queue that you want to open. To open a user's default messaging queue, just specify the user record of that user. To open a queue that you created, use the same `OCERecipient` structure that you used to create the queue.

`identity` The authentication identity of the opener of the queue. If the physical queue is on a server computer, only the server administrator and the owner of the physical queue can open a new virtual queue.

`filter` A pointer to the message filter for this virtual queue.

Set this field to `nil` if you do not want the IPM Manager to associate any filter with this queue.

`newQueueRef`	The reference number for the queue. You must use this reference number when you change the queue filter or list, open, close, or delete messages.
`notificationProc`	Reserved. You must set this field to `nil`.

DESCRIPTION

The `IPMOpenQueue` function opens the message queue you specify, creating a virtual queue with the message filter you provide. The function returns a reference number that uniquely identifies this virtual queue. When you call this function, you must specify a message-context reference number. The context links together several queues so that you can simultaneously close them simply by closing the context. If you have not already created the message context to which you want this queue to belong, you must call the `IPMOpenContext` function before calling the `IPMOpenQueue` function. You can open the same physical queue any number of times, creating a new virtual queue each time.

You specify a virtual-queue reference number whenever you list or open messages. Once you have opened a message, you must provide the same queue reference number when you call the `IPMCloseMsg` function or the `IPMCloseQueue` function. If you call the `IPMCloseQueue` function, the IPM Manager simultaneously closes all the messages that you opened with that queue reference number. If you call the `IPMCloseContext` function, the IPM Manager simultaneously closes all the messages associated with all the queues that belong to that context, and closes all of those queues.

The message filter determines which messages in the physical queue are listed by the `IPMEnumerateQueue` function when you provide the reference number for this virtual queue, which messages you can open through the queue, and which messages you can close and delete through the queue. For example, you can open a virtual queue for the default input queue with a filter that passes only high-priority messages. Then, when you call the `IPMOpenMsg` function with that queue reference number, the function allows you to open only the high-priority messages in the default input queue. If you do not provide a filter for the queue, these functions operate on all the messages in the physical queue.

SPECIAL CONSIDERATIONS

Although you allocate the pointer to the queue filter, the IPM Manager owns the pointer until you close the queue or call the `IPMChangeQueueFilter` function to replace the filter. Do not reuse or dispose of this pointer until you close the queue or replace the filter.

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0409

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidFilter`	–15105	The specified filter is invalid
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed
`kIPMBadQName`	–15112	Invalid queue name
`kIPMBadContext`	–15118	Invalid context reference
`kIPMContextIsClosing`	–15119	The specified context is closing

SEE ALSO

To create a new queue before opening it, use the `IPMCreateQueue` function (page 7-69).

You can change the queue filter by calling the `IPMChangeQueueFilter` function, described next. See "Filter Structures" on page 7-34 for information on queue filters.

Call the `IPMCloseQueue` function (page 7-76) to close a virtual queue.

IPMChangeQueueFilter

The `IPMChangeQueueFilter` function sets a new filter for a specific virtual queue.

```
pascal OSErr IPMChangeQueueFilter(IPMParamBlockPtr paramBlock,
                                  Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async` A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`queueRef`	`IPMQueueRef`	Virtual-queue reference number.
↔	`filter`	`IPMFilter*`	Pointer to the queue filter.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`queueRef`	The virtual-queue reference number returned by the `IPMOpenQueue` function. This number identifies the virtual queue to which the request applies.
`filter`	A pointer to an `IPMFilter` structure that specifies the new filter that you want the IPM Manager to apply to the queue. Set this field to `nil` to remove all filters from this queue. When the `IPMChangeQueueFilter` function completes execution, it returns a pointer to the filter that was in effect when you called the function. The IPM Manager has no further use for this pointer, and you can now dispose of it.

DESCRIPTION

The `IPMChangeQueueFilter` function applies the filter specified in the `filter` parameter to the virtual queue indicated by the `queueRef` parameter. If you set the `filter` parameter to `nil`, the function sets the filter for the virtual queue to the default filter, which matches all messages in the physical queue.

SPECIAL CONSIDERATIONS

Although you allocate the pointer to the queue filter, the IPM Manager owns the pointer until you close the queue or call the `IPMChangeQueueFilter` function to replace the filter. Do not reuse or dispose of this pointer until you close the queue or replace the filter.

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0414

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidMsgType`	–15091	Message type is invalid
`kIPMInvalidFilter`	–15105	Filter is invalid
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

See “Filter Structures” on page 7-34 for information on queue filters.

You set the queue filter initially when you open the queue; see the description of the `IPMOpenQueue` function on page 7-72.

IPMCloseQueue

The `IPMCloseQueue` function closes the specified virtual message queue.

```
pascal OSErr IPMCloseQueue(IPMParamBlockPtr paramBlock,
                           Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`queueRef`	`IPMQueueRef`	Virtual-queue reference number.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`queueRef`	The virtual-queue reference number returned by the `IPMOpenQueue` function. This number identifies the virtual queue you wish to close.

DESCRIPTION

You can call the `IPMCloseQueue` function at any time that the specified virtual queue is open. When you call this function, the function first closes any messages that you opened using the queue reference number for this queue. The function then closes the virtual queue and disassociates the queue from its context.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$040A

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter

SEE ALSO

You use the `IPMOpenQueue` function (page 7-72) to open a virtual queue.

You can use the `IPMCloseMsg` function (page 7-104) to close an individual message.

You can use the `IPMCloseContext` function, described next, to close simultaneously all of the queues associated with a specific context.

You can use the `IPMDeleteQueue` function (page 7-78) to delete a physical queue after you have closed all of its associated virtual queues.

IPMCloseContext

The `IPMCloseContext` function closes all of the messages and queues that are associated with the specified context and then eliminates that context.

```
pascal OSErr IPMCloseContext(IPMParamBlockPtr paramBlock,
                             Boolean async);
```

`paramBlock`
: A pointer to a parameter block.

`async`
: A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`contextRef`	`IPMQueueRef`	Context reference number.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`contextRef`
: The context reference number returned by the `IPMOpenContext` function. This number identifies the context you wish to close.

DESCRIPTION

When you open a virtual queue, you provide a context reference number that specifies the context to which that queue belongs. When you close a context, the `IPMCloseContext` function first closes all of the messages that you opened for the queues that belong to that context. Next, it closes all of the queues that belong to the context, and finally, it eliminates the context itself, so that the context reference number is no longer valid.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_oceTBDispatch`	$0401

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMBadContext`	–15118	Invalid context reference
`kIPMContextIsClosing`	–15119	The specified context is already closed

SEE ALSO

You use the `IPMOpenContext` function (page 7-70) to create a context.

You use the `IPMOpenQueue` function (page 7-72) to open a queue and associate it with a specific context.

You can use the `IPMCloseMsg` function (page 7-104) to close a specific message and the `IPMCloseQueue` function (page 7-76) to close a specific queue.

IPMDeleteQueue

The `IPMDeleteQueue` function deletes the specified physical message queue.

```
pascal OSErr IPMDeleteQueue(IPMParamBlockPtr paramBlock,
                            Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async`
A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`queue`	`OCERecipient*`	Queue that you want to delete.
→	`identity`	`AuthIdentity`	Authentication identity.
→	`owner`	`PackedRecordID*`	Owner of the queue.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`queue`	A pointer to an `OCERecipient` structure that specifies the name and location of the queue that you want to delete.
`identity`	The authentication identity of the owner of the queue or of the server administrator if this queue is on a server computer. The IPM Manager ignores this field if the queue is on the local computer.
`owner`	A pointer to the packed record ID of the owner of the queue. If the queue is on a remote computer, you must specify the owner of the queue in this field. The IPM Manager ignores this field if the queue is on the local computer.

DESCRIPTION

Before you can delete a physical queue, you must close any open virtual queues associated with that physical queue. You can delete a queue at any time that the queue is not open, provided it is on the local computer or, if it is on a server computer, you have the appropriate access privileges. The AOCE server allows only the server administrator to delete a queue.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0412

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMBadQName`	–15112	Invalid queue name
`kIPMQBusy`	–15126	Queue busy; cannot delete

SEE ALSO

You use the `IPMCreateQueue` function (page 7-69) to create a physical queue and the `IPMOpenQueue` function (page 7-72) to open a virtual queue.

You use the `IPMCloseQueue` function (page 7-76) to close a virtual queue.

Listing and Reading Messages

A queue can contain any number of messages. This section describes the functions you can use to list the messages in a message queue, open a message or a nested-message block, read a message header and message blocks, and close a message.

IPMEnumerateQueue

The `IPMEnumerateQueue` function returns a list of messages in the specified queue that match the filter criteria that you provide in the function.

```
pascal OSErr IPMEnumerateQueue(IPMParamBlockPtr paramBlock,
                               Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async` A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`queueRef`	`IPMQueueRef`	Queue reference number.
→	`startSeqNum`	`IPMSeqNum`	First message to list.
→	`getProcHint`	`Boolean`	List process hints?
→	`getMsgType`	`Boolean`	List message types?
→	`filter`	`IPMFilter*`	Pointer to queue filter.
→	`numToGet`	`unsigned short`	Number of messages to list.
←	`numGotten`	`unsigned short`	Number of messages listed.
→	`enumCount`	`unsigned long`	Buffer size.
→	`enumBuffer`	`Ptr`	Pointer to buffer.
←	`actEnumCount`	`unsigned long`	Number of bytes returned in buffer.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`queueRef`	A pointer to an `OCERecipient` structure that specifies the name and location of the virtual queue that you want to enumerate.
`startSeqNum`	The sequence number of the first message for which you want the function to return information. Sequence numbers start with 1.
`getProcHint`	A Boolean value that indicates whether you want the function to include the process hint of each listed message. You can specify a process hint for a message when you call the `IPMNewMsg`, `IPMNewHFSMsg`, or `IPMNewNestedMsgBlock` function to start the message.
`getMsgType`	A Boolean value that indicates whether you want the function to include the message type of each listed message.
`filter`	A pointer to the filter to use for this enumeration of the queue. If you provide a valid pointer to a filter, the function uses it only for this enumeration; the current filter for this virtual queue remains in effect after the function completes execution. (The current filter is the one you specified most recently with the `IPMOpenQueue` or `IPMChangeQueueFilter` function.) Set the `filter` field to `nil` to use the current filter. Set this field to –1 to ignore all filters and list all the messages in the physical queue.
`numToGet`	The number of messages that you want listed.
`numGotten`	The number of messages that the function actually listed in your buffer.
`enumCount`	The size, in bytes, of the buffer you are providing.
`enumBuffer`	A pointer to the buffer that you are providing.
`actEnumCount`	The number of bytes of data that the function wrote to your buffer.

DESCRIPTION

For each message in the physical input queue that matches your filter criteria, the `IPMEnumerateQueue` function places a structure of type `IPMMsgInfo` in your buffer. You must allocate a buffer large enough to hold at least one complete `IPMMsgInfo` structure. The last two fields in this structure, `procHint` and `msgType`, are present only if you specify `true` for the `getProcHint` and `getMsgType` parameters of the `IPMEnumerateQueue` function. Both the `procHint` and `msgType` fields, if present, are packed structures and can be anywhere from 0 to 33 bytes in size.

You can use the `numToGet` parameter to specify the total number of messages you want listed. In the `numGotten` parameter, the function returns the actual number of messages listed and, in the `actEnumCount` parameter, the number of bytes it wrote to your buffer. The function does not return partial `IPMMsgInfo` structures.

The first time you call the `IPMEnumerateQueue` function to list the messages in a queue, specify 1 for the `startSeqNum` parameter. If the function returns information for as many messages as you requested in the `numToGet` parameter or puts as many

`IPMMsgInfo` structures in your buffer as the buffer will hold, you can assume that the queue holds more messages to be listed. In this case, increment the number in the `startSeqNum` parameter by the number of messages listed (that is, by the number returned in the `numGotten` parameter) and call the function again.

Note
Do not call the `IPMEnumerateQueue` function any more often than necessary; every user connected to a server periodically requests a list of messages, and a server's overall performance can be noticeably affected if it has to process too many enumeration requests. ◆

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_oceTBDispatch`	$0413

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidMsgType`	–15091	Message type is invalid
`kIPMInvalidFilter`	–15105	Filter is invalid
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed
`kIPMeoQ`	–15120	No more messages

SEE ALSO

The `IPMEnumerateQueue` function places structures of type `IPMMsgInfo` in your buffer. The `IPMMsgInfo` data type is described in "Message Information Structure" on page 7-36.

IPMOpenMsg

The `IPMOpenMsg` function opens the specified message in the specified queue.

```
pascal OSErr IPMOpenMsg(IPMParamBlockPtr paramBlock,
                        Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`queueRef`	`IPMQueueRef`	Queue reference number.
→	`sequenceNum`	`IPMSeqNum`	Message sequence number requested.
←	`newMsgRef`	`IPMMsgRef`	Message reference number.
←	`actualSeqNum`	`IPMSeqNum`	Sequence number of message actually opened.
→	`exactMatch`	`Boolean`	Match requested sequence number exactly?

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`queueRef`	The queue reference number of the virtual queue containing the message that you want to open.
`sequenceNum`	The sequence number of the message you wish to open, or, if you set the `exactMatch` field to `false`, the sequence number at which you want the function to start looking for a message to open. Sequence numbers start with 1.
`newMsgRef`	The message reference number of the opened message. You must use this number when you call the `IPMVerifySignature` function to verify a signature, when you call the `IPMCloseMsg` function to close the message, and any time you read information from the message.
`actualSeqNum`	The actual sequence number of the message opened by the function. This value always equals the number you specify in the `sequenceNum` field unless you set the `exactMatch` field to `false`, in which case the message opened might have a sequence number higher than the one you requested.
`exactMatch`	A Boolean value that specifies whether the sequence number of the message opened must be exactly the same as the number you specify in the `sequenceNum` field. If you set the `exactMatch` field to `false`, the function opens the next message that has a sequence number equal to or greater than the one you specify in the `sequenceNum` field and that passes the current filter criteria for the queue.

DESCRIPTION

You must call the `IPMOpenMsg` function before you can read any of the information in a message in a message queue.

The IPM Manager assigns a sequence number to each message in a physical queue when it adds that message to the queue. Because the IPM Manager does not reuse the number of a message that is removed from the queue, some sequence numbers might be missing from the queue.

The `IPMOpenMsg` function opens a message only if it meets the current filter criteria for the virtual queue. If you specify a message sequence number for a message that does not meet the filter criteria and set the `exactMatch` field to `true`, the `IPMOpenMsg` function returns the `kIPMEltNotFound` result code.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$040B

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

You can use the `IPMEnumerateQueue` function (page 7-80) to list the messages in a queue.

Use the `IPMOpenHFSMsg` function, described next, to open a message on disk.

Use the `IPMOpenBlockAsMsg` function (page 7-86) to open a nested message.

IPMOpenHFSMsg

The `IPMOpenHFSMsg` function opens the specified HFS file as a message.

```
pascal OSErr IPMOpenHFSMsg(IPMParamBlockPtr paramBlock,
                            Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async` A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`hfsPath`	`FSSpec*`	Specifier of the file to open.
←	`newMsgRef`	`IPMMsgRef`	Message reference number.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`hfsPath`	The file system specification structure for the file you wish to open as a message.
`newMsgRef`	The message reference number of the opened message. You must use this number when you read information from the message, when you call the `IPMVerifySignature` function to verify a signature, or when you call the `IPMCloseMsg` function to close the message.

DESCRIPTION

You must call the `IPMOpenHFSMsg` function before you can read any of the information in a message that is in an HFS file on disk.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0417

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

Use the `IPMOpenMsg` function (page 7-82) to open a message in a message queue.

Use the `IPMOpenBlockAsMsg` function, described next, to open a nested message.

IPMOpenBlockAsMsg

The `IPMOpenBlockAsMsg` function opens a nested message.

```
pascal OSErr IPMOpenBlockAsMsg(IPMParamBlockPtr paramBlock,
                               Boolean async);
```

`paramBlock`
: A pointer to a parameter block.

`async`
: A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number of the enclosing message.
←	`newMsgRef`	`IPMMsgRef`	Message reference number of the nested message.
→	`blockIndex`	`unsigned short`	Index value of block containing nested message.

See "Interprogram Messaging Parameter Block Header," beginning on page 7-40, for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`
: The message reference number of the message that contains the nested message you want to read. This number is returned by the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function when you open the containing message.

`newMsgRef`
: The message reference number of the opened nested message. You must use this number when you read information from the message, when you call the `IPMVerifySignature` function to verify a signature, or when you call the `IPMCloseMsg` function to close the message.

`blockIndex`
: The sequential position of the block that you want to open as a message. For example, if you want to open the tenth block, you set `blockIndex` to 10. You can use the `IPMGetBlkIndex` function to get the index number of a block.

DESCRIPTION

The `IPMOpenBlockAsMsg` function opens a nested message so that you can use other IPM Manager functions to read information from it. Before you use this function, you must open the containing message (which can also be a nested message), and you must know the index number of the nested-message block within the containing message. A nested message has a creator type of `kIPMSignature` and a block type of `kIPMEnclosedMsgType`.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$040F

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMBlockIsNotNestedMsg`	–15101	Block is not message
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

Use the `IPMGetBlkIndex` function (page 7-96) to get the index number of a block.

IPMGetMsgInfo

The `IPMGetMsgInfo` function returns information about a message in a message queue.

```
pascal OSErr IPMGetMsgInfo(IPMParamBlockPtr paramBlock,
                           Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
↔	`info`	`IPMMsgInfo*`	Pointer to returned information.

See "Interprogram Messaging Parameter Block Header," beginning on page 7-40, for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message about which you want information. This number is returned by the `IPMOpenMsg` function when you open the message.
`info`	A pointer to an `IPMMsgInfo` structure in which the function returns information about the message. You must allocate this structure. The function always returns the full `IPMGetMsgInfo` structure, which is of variable length and packed; the maximum size of this structure is 130 bytes.

DESCRIPTION

You can call the `IPMGetMsgInfo` function after you open a message in a queue. You cannot use the `IPMGetMsgInfo` function to obtain information about a message stored in a file on disk or to get information about a nested message.

The `IPMGetMsgInfo` function returns the same information about a message as the `IPMEnumerateQueue` function returns.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0419

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

The `IPMGetMsgInfo` function returns the same information about a message as the `IPMEnumerateQueue` function (page 7-80) returns.

The `IPMGetMsgInfo` function returns information in an `IPMGetMsgInfo` structure, described in "Message Information Structure" on page 7-36.

Use the `IPMReadHeader` function, described next, to obtain header information from nested messages and messages stored on disk, or to get information from header fields not returned by the `IPMGetMsgInfo` function.

IPMReadHeader

The `IPMReadHeader` function reads the contents of a specified header field of a message.

```
pascal OSErr IPMReadHeader(IPMParamBlockPtr paramBlock,
                           Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`fieldSelector`	`unsigned short`	Message header field selector.
→	`offset`	`long`	Offset to header field.
→	`count`	`unsigned long`	The size, in bytes, of the output buffer.
→	`buffer`	`Ptr`	Pointer to your buffer.
←	`actualCount`	`unsigned long`	Number of bytes of data read.
←	`remaining`	`unsigned long`	Number of bytes of data remaining to be read.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message whose header you want to read. This number is returned by the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function when you open the message.
`fieldSelector`	The message-header field or fields that you want to read. You can set the `fieldSelector` field to the values shown in the description section that follows.

`offset`	The offset to the header field at which you want to start reading. Set this field to 0 to start reading a header field at the beginning. If the `IPMReadHeader` function returns a value in the `remaining` field, you can increment the value in the `offset` field by the value returned in the `actualCount` field and call the function again to continue reading from the header field.
`count`	The size, in bytes, of the buffer you provide.
`buffer`	A pointer to your buffer.
`actualCount`	The number of bytes of data actually written to your buffer.
`remaining`	The number of bytes of data in this header field remaining to be read.

DESCRIPTION

The `IPMReadHeader` function returns information about one or more fields of a message header. If the buffer you provide is not large enough to hold all the data you request, the function returns, in the `remaining` parameter, the number of bytes remaining. You can then increment the value in the `offset` parameter by the value in the `actualCount` parameter and call the function again. You must open the message with the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function before you can call the `IPMReadHeader` function.

Use the `fieldSelector` parameter to indicate the field of the message header that you want to read. You can set this parameter to any of the following values:

```
enum {
   kIPMTOC = 0,
   kIPMSender = 1,
   kIPMProcessHint = 2,
   kIPMMessageTitle = 3,
   kIPMMessageType = 4,
   kIPMFixedInfo = 7
};

typedef Byte IPMHeaderSelector;
```

Constant descriptions

`kIPMTOC`	The message table of contents (TOC). The TOC contains information about each block in the message. The `IPMReadHeader` function returns an array of `IPMTOC` structures, each containing information about one block. The `IPMTOC` structure is described on page 7-37.

`kIPMSender`	The sender of the message, in an `IPMSender` structure. If the message is authenticated, the IPM Manager fills in this field from the identity of the originator of the message, and this field provides the authenticated originator of the message. If the message is not authenticated, the creator of the message specifies the contents of this field. The `IPMSender` structure is described on page 7-40. The `IPMFixedHdrInfo` structure (page 7-38) includes an `authenticated` field.
`kIPMProcessHint`	The process hint of the message, which is a Pascal string of up to 32 characters. The value of meaning of the process hint is defined by the creator of the message.
`kIPMMessageTitle`	The message title. This title is specified by the creator of the message and normally indicates the subject, purpose, or content of the message.
`kIPMMessageType`	The message type, in an `IPMMsgType` structure (page 7-28).
`kIPMFixedInfo`	A standard subset of the fields in the header, in an `IPMFixedHdrInfo` structure (page 7-38). When you set the `fieldSelector` parameter to `kIPMFixedInfo`, the IPM Manager ignores the `offset` and `count` fields.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$040E

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidOffset`	–15093	Bad offset for read or write operation
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

The `IPMSender` structure is described in “Sender Structure” on page 7-39.

The `IPMTOC` structure is described on page 7-37.

The `IPMMsgType` structure is described on page 7-28.

The `IPMFixedHdrInfo` structure is described on page 7-38.

IPMReadRecipient

The `IPMReadRecipient` function reads a recipient from a message header.

```
pascal OSErr IPMReadRecipient(IPMParamBlockPtr paramBlock,
                              Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async` A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`rcptIndex`	`unsigned short`	Recipient index number.
→	`offset`	`long`	Offset to recipient data.
→	`count`	`unsigned long`	Buffer size.
→	`buffer`	`Ptr`	Pointer to your buffer.
←	`actualCount`	`unsigned long`	Number of bytes of data read.
→	`reserved`	`short`	Must be 0.
←	`remaining`	`unsigned long`	Number of bytes of data remaining to be read.
←	`originalIndex`	`unsigned short`	Original recipient index.
←	`OCERecipientOffsetFlags`	`recipientOffsetFlags`	Recipient-type flags.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef` The message reference number of the message whose recipient data you want to read. This number is returned by the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function when you open the message.

`rcptIndex` The index number of the recipient you want to read. Recipient index numbers are sequential, starting with 1.

`offset`	The offset to the data for the specified recipient at which to start reading. The first time you call the `IPMReadRecipient` function for a given recipient you should set this field to 0. If your buffer is not large enough to hold all of the recipient data, you can increment the value in the `offset` field by the value returned in the `actualCount` field and call the function again.
`count`	The size, in bytes, of your buffer.
`buffer`	A pointer to your buffer. The function places the information about the recipient in your buffer in the form of an `OCEPackedRecipient` structure.
`actualCount`	The number of bytes of data the function wrote to your buffer.
`reserved`	Reserved; you must set this field to 0.
`remaining`	The number of bytes of data remaining to be read. If this field returns a nonzero value, you should increment the value in the `offset` field by the value returned in the `actualCount` field and call the function again.
`originalIndex`	The index of this recipient in the original recipient list (that is, the recipient list before the IPM Manager resolves any group addresses).
`OCERecipientOffsetFlags`	A flag byte that provides information about the recipient.

DESCRIPTION

The `IPMReadRecipient` function returns recipient information from the header of a message. If the original message header contained recipient addresses that were groups or that identified records containing the address of the actual recipient, the `IPMReadRecipient` function returns the final recipients of the message.

The `OCERecipientOffsetFlags` field contains the following bits:

```
enum {
   kIPMFromDistListBit = 0,
   kIPMDummyRecBit = 1,
   kIPMFeedbackRecBit = 2,
   kIPMReporterRecBit = 3,
   kIPMBCCRecBit = 4
};
```

Flag descriptions

`kIPMFromDistListBit`	Reserved.
`kIPMDummyRecBit`	If this flag is set to 1, the IPM Manager delivered the message to this recipient.
`kIPMFeedbackRecBit`	Reserved.

`kIPMReporterRecBit`	Reserved.
`kIPMBCCRecBit`	If this flag is set to 1, this is a "bcc" (blind carbon copy) recipient; in other words, this recipient is not included in the recipient list received by the other recipients of the message. You can receive this flag only if you sent the letter or if you were the bcc recipient.

You can use the following mask values to test these flags:

```
enum {
   kIPMFromDistListMask=   1<<kIPMFromDistListBit,
   kIPMDummyRecMask=       1<<kIPMDummyRecBit,
   kIPMFeedbackRecMask=    1<<kIPMFeedbackRecBit,
   kIPMReporterRecMask=    1<<kIPMReporterRecBit,
   kIPMBCCRecMask=         1<<kIPMBCCRecBit
};
```

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0410

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidOffset`	–15093	Bad offset for read or write operation
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

The `IPMReadRecipient` function places the information about the recipient in your buffer in the form of an `OCEPackedRecipient` structure (page 7-25).

IPMReadReplyQueue

The `IPMReadReplyQueue` function reads the reply queue field of the message header.

```
pascal OSErr IPMReadReplyQueue(IPMParamBlockPtr paramBlock,
                               Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`offset`	`long`	Offset to reply queue data.
→	`count`	`unsigned long`	Buffer size.
→	`buffer`	`Ptr`	Pointer to your buffer.
←	`actualCount`	`unsigned long`	Number of bytes of data read.
→	`reserved`	`short`	Must be 0.
←	`remaining`	`unsigned long`	Number of bytes of data remaining to be read.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message whose reply queue data you want to read. This number is returned by the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function when you open the message.
`offset`	The offset to the data at which to start reading. The first time you call the `IPMReadReplyQueue` function, you should set this value to 0. If your buffer is not large enough to hold all of the reply queue data, you can increment the value in the `offset` field by the value returned in the `actualCount` field and call the function again.
`count`	The size, in bytes, of your buffer.
`buffer`	A pointer to your buffer. The function places the information about the reply queue in your buffer in the form of an `OCEPackedRecipient` structure.
`actualCount`	The number of bytes of data the function wrote to your buffer.
`reserved`	Reserved; you must set this field to 0.
`remaining`	The number of bytes of data remaining to be read. If this field returns a nonzero value, you should increment the value in the `offset` field by the value returned in the `actualCount` field and call the function again.

DESCRIPTION

The reply queue is the address to which the IPM Manager returns delivery and nondelivery reports and to which you should address reply messages.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_oceTBDispatch`	$0421

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidOffset`	–15093	Bad offset for read or write operation
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMAttrNotInHdr`	–15106	No reply queue in message header
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

The `IPMReadReplyQueue` function places the information about the reply queue in your buffer in the form of an `OCEPackedRecipient` structure (page 7-25).

IPMGetBlkIndex

The `IPMGetBlkIndex` function returns the block type and index value for the first block encountered that matches the specifications you provide.

```
pascal OSErr IPMGetBlkIndex(IPMParamBlockPtr paramBlock,
                            Boolean async);
```

`paramBlock`
A pointer to a parameter block.

`async`
A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`blockType`	`IPMBlockType`	Block types to return.
→	`index`	`unsigned short`	Number of matches to find before returning information.
→	`startingFrom`	`unsigned short`	Starting index.
←	`actualBlockType`	`IPMBlockType`	Block type of block returned.
←	`actualBlockIndex`	`unsigned short`	Index value of block returned.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message from which you want information. This number is returned by the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function when you open the message.
`blockType`	The creator and type of the block for which you want an index value. You can use the `kIPMTypeWildCard` wildcard value for the creator field, the type field, or both.
`index`	The number of matches the function should find before it returns the index and type of a block. For example, if you set the `index` field to 5, the function returns the index and type of the fifth block it finds that matches the value you specify in the `blockType` field.
`startingFrom`	The index number of the block at which to begin the search. Index numbers start at 1.
`actualBlockType`	The creator and type of the block that matches all of your search criteria.
`actualBlockIndex`	The index number of the block that matches all of your search criteria.

DESCRIPTION

Each IPM message can contain message blocks. You can use the `IPMGetBlkIndex` function to determine the type and creator of each block, or the sequential position (referred to as the *index number*) of blocks that have specific types.

If you want to get information about every block in the message, you can specify the wildcard value `kIPMTypeWildCard` for the creator and type and call the function repeatedly, incrementing the value in the `startingFrom` field each time. If you want to get information about every block of a specific type or with a specific creator, put that type or creator in the `blockType` field and call the function repeatedly, incrementing the value in the `index` field each time.

If the function does not find any more matches to your criteria, it returns the `kOCEInvalidIndex` result code.

You can use the value returned in the `actualBlockIndex` field to identify a block you want to read when you call the `IPMReadMsg` function.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0418

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMBlkNotFound`	–15107	Specified block nonexistent
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

To read a message block, call the `IPMReadMsg` function, described next.

IPMReadMsg

The `IPMReadMsg` function reads data from an IPM message.

```
pascal OSErr IPMReadMsg(IPMParamBlockPtr paramBlock,
                        Boolean async);
```

`paramBlock`
: A pointer to a parameter block.

`async`
: A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`mode`	`IPMAccessMode`	Mode in which the offset should be interpreted.
→	`offset`	`long`	Offset to the starting point of the read.
→	`count`	`unsigned long`	Buffer size.
→	`buffer`	`Ptr`	Pointer to your buffer.
←	`actualCount`	`unsigned long`	Number of bytes of data read.
→	`blockIndex`	`unsigned short`	Index number of the block to read.
←	`remaining`	`unsigned long`	Number of bytes of data remaining to be read.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message you want to read. This number is returned by the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function when you open the message.
`mode`	The mode in which the offset parameter is to be interpreted. The function uses this field to determine whether to begin reading data relative to the end of the last data read, to the beginning of the block, or to the end of the block. See the discussion following these field descriptions for details.
`offset`	An offset that the function uses when it calculates the starting point of the read operation. Set this value to 0 when you start reading a block from the beginning. See the following discussion for details.
`count`	The size, in bytes, of the buffer that you are providing.
`buffer`	A pointer to your buffer.
`actualCount`	The number of bytes of data actually written to your buffer.
`blockIndex`	The sequential position of the block that you want to read. For example, if you want to read the tenth block, you set `blockIndex` to 10. You can use the `IPMGetBlkIndex` function to get the creator, block type, and index number of a block. If you set the `blockIndex` field to 0, the `IPMReadMsg` function treats all the blocks in the message, including the message header, as a single unit, ignoring all block boundaries.
`remaining`	The number of bytes of data remaining to be read. If this field returns a nonzero value, you can increment the value in the `offset` field by the value in the `actualCount` field and call the `IPMReadMsg` function again to read the next portion of data.

DESCRIPTION

The `IPMReadMsg` function can treat the entire message body as a single unit (if you set the `blockIndex` parameter to 0) or can read a specific message block.

The IPM Manager uses a marker (referred to as the *message mark*) that points to the current location within a message that you are reading. After the `IPMReadMsg` function completes, the message mark points to the byte following the last byte read.

You use the `mode` and `offset` parameters to specify the point in the message at which the `IPMReadMsg` function starts reading. The `mode` parameter indicates whether you want the `IPMReadMsg` function to begin reading at the current position of the mark or to calculate another starting point relative to the beginning of the message, the beginning of the block, the end of the message, or the current mark location. You can set the `mode` parameter to any one of the following values:

```
enum {
   kIPMAtMark,
   kIPMFromStart,
   kIPMFromLEOM,
   kIPMFromMark
};
```

Constant descriptions

`kIPMAtMark`	The `IPMReadMsg` function starts reading at the current position of the mark. In this case, the function ignores the offset value. This mode is useful, for example, for reading in sequence through a block.
`kIPMFromStart`	The function interprets the value in the `offset` parameter as an offset from the beginning of the block you specify by the `blockIndex` parameter. If you specify 0 for the `blockIndex` parameter, the function interprets the value in the `offset` parameter as an offset from the beginning of the message body. If you want to start reading at the 100th byte of the second block in the message, for example, set the `blockIndex` parameter to 2, the `mode` parameter to `kIPMFromStart`, and the `offset` parameter to 100. When you use this mode, you cannot set the `offset` parameter to a negative value or you will be reading data that is not part of the message.
`kIPMFromLEOM`	The function interprets the value in the `offset` parameter as an offset from the end of the block you specify by the `blockIndex` parameter. If you specify 0 for the `blockIndex` parameter, the function interprets the value in the `offset` parameter as an offset from the end of the message. When you use this mode, the `offset` parameter must be a negative value or you will be reading data that is not part of the message.

`kIPMFromMark`	The function interprets the value in the `offset` parameter as an offset from the current position of the mark. Use a negative offset value to indicate a starting point prior to the current position of the mark and a positive offset value to indicate a starting point following the current position of the mark.

A message block that has a creator type of `kIPMSignature` and a block type of `kIPMEnclosedMsgType` contains a nested message. To read the contents of such a block, first use the `IPMOpenBlockAsMsg` function to open the nested message and then use the other functions in this section to read its contents.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_oceTBDispatch`	$040D

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMInvalidOffset`	–15093	Bad offset for read or write operation
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMCorruptDataStructures`	–15099	Message is corrupt
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

Use the `IPMGetBlkIndex` function (page 7-96) to get the creator, block type, and index number of a block.

Use the `IPMOpenBlockAsMsg` function (page 7-86) to read a block containing a nested message.

IPMVerifySignature

The `IPMVerifySignature` function verifies a digital signature for a message.

```
pascal OSErr IPMVerifySignature(IPMParamBlockPtr paramBlock);
```

`paramBlock`
A pointer to a parameter block.

Parameter block

←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`signatureContext`	`SIGContextPtr`	Signature context.

See “Interprogram Messaging Parameter Block Header” on page 7-40 for a description of the `ioResult` field.

Field descriptions

`msgRef`	The message reference number of the message from which you want information. This number is returned by the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function when you open the message.
`signatureContext`	The signature context you obtained from the `SIGNewContext` function and provided to the `SIGVerifyPrepare` function.

DESCRIPTION

If the creator of the message used the `IPMEndMsg` function to add a digital signature to the message, you can use the `IPMVerifySignature` function to verify the signature. You can use the `IPMReadHeader` function to determine whether a message has a digital signature. The `IPMEndMsg` function places the digital signature in a block with a creator of `kIPMSignature` and a type of `kIPMDigitalSignature`.

To verify a signature, use the `IPMGetBlkIndex` function to get the index number of the signature block and the `IPMReadMsg` function to read the signature into a buffer. Then call the `SIGNewContext` and `SIGVerifyPrepare` functions to begin the process of verifying the signature. When you pass a pointer to the signature context (returned by the `SIGNewContext` function) in the `signatureContext` parameter to the `IPMVerifySignature` function, the function verifies the signature.

Note

The signature context used to create a digital signature has no relationship to the contexts discussed in “Managing Message Queues” starting on page 7-68 and elsewhere in this chapter. ◆

Because the IPM Manager modifies some fields in the message header during message transmission and delivery, not all header fields can be signed. For example, the final number of recipients, resolution count, and hop count fields are not signed. All message blocks except the signature block itself are signed.

SPECIAL CONSIDERATIONS

You cannot execute the `IPMVerifySignature` function asynchronously; therefore, you can not call this function at interrupt time.

There must also be at least 8.5 KB of stack space available when you call this function.

If you are verifying a digital signature for a large message, the `IPMVerifySignature` function can take a long time to complete (up to several minutes on some computers). You should display a dialog box informing the user of this possibility.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0422

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMBlkNotFound`	–15107	Specified block nonexistent
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed

SEE ALSO

You use the `IPMEndMsg` function (page 7-65) to add a digital signature to a message.

You can use the `IPMReadHeader` function (page 7-89) to determine if a message has been signed.

Use the `IPMGetBlkIndex` function (page 7-96) to get the index number of the signature block and the `IPMReadMsg` function (page 7-98) to read the signature block.

Digital signatures and the `SIGNewContext` and `SIGVerifyPrepare` functions are discussed in the chapter "Digital Signature Manager" in this book.

IPMCloseMsg

The `IPMCloseMsg` function closes a message, invalidating the message reference number, and can delete the message.

```
pascal OSErr IPMCloseMsg(IPMParamBlockPtr paramBlock,
                         Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`msgRef`	`IPMMsgRef`	Message reference number.
→	`deleteMsg`	`Boolean`	Delete the message?

See “Interprogram Messaging Parameter Block Header” on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`msgRef`	The message reference number of the message you want to close. This number is returned by the `IPMOpenMsg`, `IPMOpenHFSMsg`, or `IPMOpenBlockAsMsg` function when you open the message.
`deleteMsg`	A Boolean value specifying whether you want to delete the message after closing it. If you set this field to `true` for a message in a message queue, the IPM Manager removes the message from the physical queue. If you set this field to `true` for a message that is an HFS file, the IPM Manager deletes the file. If the message is a nested message, the `IPMCloseMsg` function ignores this field.

DESCRIPTION

When you have finished reading information from a message, you should call the `IPMCloseMsg` function so that the IPM Manager can release the memory it allocates when you open a message. You can set the `deleteMsg` parameter to `true` to have the IPM Manager delete the message after it closes it. (The `IPMCloseMsg` function will always close a message that was opened through the queue you specify with the message reference number, but if the same message is also open through another virtual queue, the function does not delete it. In that case, the function returns the `kIPMEltBusy` result code.) If you do not delete the message, it remains in the message queue or on disk and you can open it again at any time.

After you close a message, its message reference number is no longer valid.

You can close a message containing an open nested message; however, you can't delete such a message.

When you call the `IPMCloseQueue` function to close a message queue, the function automatically closes all of the messages that you opened through that queue's reference number. When you call the `IPMCloseContext` function to close a context, it first closes all of the messages that you opened for the queues that belong to that context.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$040C

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMEltBusy`	–15116	Message is in use

SEE ALSO

You use the `IPMOpenMsg` (page 7-82), `IPMOpenHFSMsg` (page 7-84), or `IPMOpenBlockAsMsg` (page 7-86) function to open a message.

The `IPMCloseQueue` function (page 7-76) closes all the messages associated with a specific virtual queue. The `IPMCloseContext` function (page 7-77) closes all the messages associated with a context.

You can use the `IPMDeleteMsgRange` function (page 7-106) to delete one or more messages in a specific virtual queue.

Deleting Messages

You can use the `IPMDeleteMsgRange` function, described in this section, to delete one or more messages in a message queue. The `IPMCloseMsg` function (page 7-104) can delete a single message after closing it.

IPMDeleteMsgRange

The `IPMDeleteMsgRange` function deletes one or more messages from a message queue.

```
pascal OSErr IPMDeleteMsgRange(IPMParamBlockPtr paramBlock,
                               Boolean async);
```

`paramBlock`	A pointer to a parameter block.
`async`	A Boolean value that specifies whether the IPM Manager should execute the function asynchronously. Set this parameter to `true` for asynchronous execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	Pointer to a completion routine.
←	`ioResult`	`OSErr`	Result of the function.
→	`queueRef`	`IPMQueueRef`	Queue reference number.
→	`startSeqNum`	`IPMSeqNum`	The starting message sequence number.
→	`endSeqNum`	`IPMSeqNum`	The ending message sequence number.
←	`lastSeqNum`	`IPMSeqNum`	The sequence number of the next message.

See "Interprogram Messaging Parameter Block Header" on page 7-40 for descriptions of the `ioCompletion` and `ioResult` fields.

Field descriptions

`queueRef`	The virtual-queue reference number returned by the `IPMOpenQueue` function. This number identifies the virtual queue to which the request applies.
`startSeqNum`	The sequence number of the first message that you want the function to delete.
`endSeqNum`	The sequence number of the last message that you want the function to delete.
`lastSeqNum`	The sequence number of the next message that remains in the queue following the last deleted message. If the function is unable to delete all of the requested messages, this field contains the sequence number of the message that the function was attempting to delete when the error occurred.

DESCRIPTION

The `IPMDeleteMsgRange` function deletes one or more messages from the physical message queue. To be deleted, a message must match the current filter for the virtual queue you specify with the `queueRef` parameter and have a sequence number falling within the range you specify with the `startSeqNum` and `endSeqNum` parameters. Note that the sequence numbers are inclusive; for instance, if you set the `startSeqNum` parameter to 5 and the `endSeqNum` parameter to 10, messages with sequence numbers 5 and 10 (if present in the specified virtual queue) are both deleted.

If the function cannot delete a particular message for some reason, the IPM Manager cancels the function without proceeding any further. In this case, the function returns the sequence number of the message that it was attempting to delete when the error occurred and also returns a result code that indicates the error. The `IPMDeleteMsgRange` function does not delete a message if it is open through any virtual queue. If you have closed the message through the virtual queue but still receive the `kIPMEltBusy` result code, the message might be open through another virtual queue. If the message is closed but contains a nested message that is still open, the function does not delete the message and returns the `kIPMEltBusy` result code.

Once you have deleted a message, you cannot open it again.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. If you call it asynchronously, you can call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_oceTBDispatch`	$0415

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed
`kIPMEltBusy`	–15116	Message is in use

SEE ALSO

You can use the `IPMCloseMsg` function (page 7-104) to delete a single message from a queue.

Utility Functions

You can use the routines in this section to work with `OCERecipient` structures.

The functions described in this section use a different assembly-language calling sequence from the other IPM Manager routines (see page 7-43). Listing 7-2 illustrates one way to do this.

Listing 7-2 Calling an MSAM utility function from assembly language

```
_oceMessaging     OPWORD   $AA5C
   SUBQ     #2,A7                    ; make room for function result
   MOVEA    param1,-(SP)             ; push the first parameter onto stack
...                                  ; push additional parameters onto stack
   MOVEQ    asyncFlag, D0            ; move async flag into d0
   MOVE.B   D0,-(SP)                 ; push the flag (byte) onto stack
   MOVEQ    #opCode, D0              ; move op code into d0
   MOVE.W   D0,-(SP)                 ; push the op code onto stack
   _oceMessaging                     ; trap call
   MOVE.W   (SP)+, D0                ; get result code
```

OCESizePackedRecipient

The `OCESizePackedRecipient` function computes the number of bytes of memory needed to hold a packed `OCERecipient` structure.

```
pascal unsigned short OCESizePackedRecipient(
                                       const OCERecipient *rcpt);
```

`rcpt`	A pointer to an `OCERecipient` structure whose size, when packed, you want to determine.

DESCRIPTION

The `OCESizePackedRecipient` function computes the number of bytes required to hold the information contained in an `OCERecipient` structure when it is packed. The number of bytes returned by the `OCESizePackedRecipient` function includes the `dataLength` field of the `OCEPackedRecipient` structure.

SPECIAL CONSIDERATIONS

The `OCESizePackedRecipient` function does not pad the value contained in the `extensionSize` field of the `OCERecipient` structure pointed to by the `rcpt` parameter. For this reason, the `OCESizePackedRecipient` function might return an odd value rather than an even one. Therefore, you need to pad the necessary fields in the `OCERecipient` structure yourself before using it as an address for a message or before passing it to any of the IPM Manager functions that require an `OCERecipient` structure of even size.

This function does not purge or move memory.

ASSEMBLY-LANGUAGE INFORMATION

Trap	**Selector**
`_OCEMessaging`	$033E

SEE ALSO

The `OCERecipient` structure is defined on page 7-24.

The `OCEPackedRecipient` structure is defined on page 7-25.

To pack an `OCERecipient` structure, use the `OCEPackRecipient` function, described next.

OCEPackRecipient

The `OCEPackRecipient` function forms an `OCEPackedRecipient` structure from an `OCERecipient` structure.

```
pascal unsigned short OCEPackRecipient(const OCERecipient *rcpt,
                                       void* buffer);
```

`rcpt`	A pointer to the `OCERecipient` structure you want to pack.
`buffer`	A pointer to the buffer in which the packed data is placed by the `OCEPackRecipient` function. You must allocate this structure.

DESCRIPTION

The `OCEPackRecipient` function packs the contents of an `OCERecipient` structure into an `OCEPackedRecipient` structure. The `OCEPackedRecipient` structure must be large enough to contain the packed `RecordID` information and any extension value of the `OCERecipient` structure. You obtain the buffer size needed by calling the `OCESizePackedRecipient` function (page 7-108).

SPECIAL CONSIDERATIONS

This function does not purge or move memory.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_OCEMessaging`	$033F

SEE ALSO

The `OCERecipient` structure is defined on page 7-24.

The `OCEPackedRecipient` structure is defined on page 7-25.

For information on unpacking an `OCEPackedRecipient` structure, see the `OCEUnpackRecipient` function, described next.

OCEUnpackRecipient

The `OCEUnpackRecipient` function unpacks an `OCEPackedRecipient` structure.

```
pascal OSErr OCEUnpackRecipient(const void* buffer,
                                OCERecipient *rcpt,
                                RecordID *entitySpecifier);
```

`buffer`	A pointer to the `OCEPackedRecipient` structure you want to unpack.
`rcpt`	A pointer to an `OCERecipient` structure. You must allocate this structure.
`entitySpecifier`	A pointer to a `RecordID` structure. The `OCEUnpackRecipient` function extracts the record identifier information from the `OCEPackedRecipient` structure and places it in this `RecordID` structure. You must allocate this structure.

DESCRIPTION

The `OCEUnpackRecipient` function extracts the information from an `OCEPackedRecipient` structure and places it in an `OCERecipient` structure and a `RecordID` structure. The `OCEUnpackRecipient` function extracts the record identifier (if any) and places it in the `RecordID` structure, places the rest of the information in the `OCERecipient` structure, and then sets the `entitySpecifier` field of the `OCERecipient` structure to point to the `RecordID` structure. The `OCEUnpackRecipient` function returns, in the `extensionValue` field of the `OCERecipient` structure, a pointer to the extension (if any), and returns the length of that extension in the `extensionSize` field of the `OCERecipient` structure. If there is no extension, the `OCEUnpackRecipient` function sets the `extensionValue` field of the `OCERecipient` structure to `nil`.

SPECIAL CONSIDERATIONS

This function does not move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_OCEMessaging`	$0340

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter

SEE ALSO

The `OCERecipient` structure is defined on page 7-24.

The `OCEPackedRecipient` structure is defined on page 7-25.

To pack an `OCERecipient` structure, see the `OCEPackRecipient` function (page 7-109).

OCEStreamRecipient

The `OCEStreamRecipient` function converts an `OCERecipient` structure from a pointer-based structure into a stream of bytes.

```
pascal OSErr OCEStreamRecipient(const OCERecipient* rcpt,
                                OCERecipientStreamer stream,
                                long userData,
                                unsigned long* actualCount);
```

`rcpt` A pointer to the `OCERecipient` structure you want to process.

`stream` A pointer to a stream function that you supply.

`userData` Data supplied by you that is passed to your stream function. The `userData` parameter can contain anything your particular stream method needs.

`actualCount`
A pointer to the total number of bytes (streamed out) by the `OCEStreamRecipient` function.

DESCRIPTION

The `OCEStreamRecipient` function converts an `OCERecipient` structure into a stream of bytes by calling the stream function that you provide. You can use this function anytime that you want to write the contents of an `OCERecipient` structure as a series of bytes to a file, into a buffer in memory, or any other place.

The stream function that you provide contains the specific code that writes out the data. The `OCEStreamRecipient` function calls your recipient stream function repeatedly and passes your function the current portion of the data that needs to be streamed, the length of this data, an `eof` flag that is set by the `OCEStreamRecipient` function if this is the last of the data to be streamed, and a `userData` parameter containing any application-specific data that you define. For example, if you were writing a stream function that wrote out an `OCERecipient` structure to a file on a hard disk, you might want to store a pointer in the `userData` parameter to a block of data that contains such information as the filename and size of the file.

If your stream function sends the `OCEStreamRecipient` function an error in the valid range for AOCE error codes, `OCEStreamRecipient` halts execution and returns the error as its result code.

SPECIAL CONSIDERATIONS

This function does not move or purge memory. However, it calls the recipient stream function that you supply in the `stream` parameter, and the stream function could move memory.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	**Selector**
`_OCEMessaging`	$0341

RESULT CODES

`noErr`	0	No error
`kOCEParamErr`	–50	Invalid parameter

SEE ALSO

The `OCERecipient` structure is defined on page 7-24.

OCESetRecipientType

Given a creation ID, the `OCESetRecipientType` function sets the extension type of an `OCERecipient` structure.

```
pascal void OCESetRecipientType(OSType extensionType,
                                CreationID *cid);
```

`extensionType`
: The type you wish to specify in an `OCERecipient` structure's `extensionType` field.

`cid` A pointer to a `CreationID` structure identifying a record. The `OCERecipient` structure for that record is the one modified by this function.

DESCRIPTION

The `OCESetRecipientType` function sets an `OCERecipient` structure's `extensionType` field to the value in the `extensionType` parameter. The `OCERecipient` is determined from the specified `cid` parameter.

If the `extensionType` field has a value of `'entn'`, then the `cid` parameter is assumed to be a valid extension and is not modified. If the `extensionType` field's value is anything else besides `'entn'`, then this routine sets the `CreationID` structure's `source` field to 0.

SPECIAL CONSIDERATIONS

The `OCESetRecipientType` function does not check whether the `cid` pointer is set to `nil`. Calling this function with the `cid` parameter set to `nil` has an indeterminate but harmful result.

This function does not move or purge memory.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_OCEMessaging`	$0343

SEE ALSO

The `OCERecipient` structure is defined on page 7-24.

To get the extension type of an `OCERecipient` structure, see the `OCEGetRecipientType` function, described next.

OCEGetRecipientType

Given a creation ID, the `OCEGetRecipientType` function returns the extension type of an `OCERecipient` structure.

```
pascal OSType OCEGetRecipientType(const CreationID *cid);
```

`cid` A pointer to a `CreationID` structure identifying a record. The `OCERecipient` structure for that record is the one read by this function.

DESCRIPTION

If you used the `OCESetRecipientType` function (page 7-112) to set the extension type of an `OCERecipient` structure, you can use the `OCEGetExtensionType` function to read the extension type.

SPECIAL CONSIDERATIONS

This function does not purge or move memory.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
`_OCEMessaging`	$0342

SEE ALSO

The `OCERecipient` structure is defined on page 7-24.

The `CreationID` structure is defined in the chapter "AOCE Utilities" in this book.

To set the extension type of an `OCERecipient` structure, see the `OCESetRecipientType` function (page 7-112).

Application-Defined Functions

This section describes routines that you can provide to be called by the IPM Manager in specific circumstances. The `MyCompletionRoutine` function is a completion routine called when an IPM Manager routine that you call asynchronously completes execution. The `MyRecipientStreamer` function is a stream-processing function that you supply to the `OCEStreamRecipient` function.

MyCompletionRoutine

When you call an IPM Manager function asynchronously, you can provide a pointer to a completion routine.

```
pascal void MyCompletionRoutine (Ptr paramBlk);
```

`paramBlk`	A pointer to the parameter block you used when you called the IPM Manager function.

Parameter block

→	`ioResult`	`OSErr`	Result of the function.

Other fields returned depend on the function that called the completion routine; see the other function descriptions in this chapter for details.

DESCRIPTION

When the IPM Manager function you called asynchronously completes execution, it calls your completion routine. Your completion routine can check the function result plus any parameters returned by the function and take appropriate action.

SPECIAL CONSIDERATIONS

The IPM Manager saves the value of your A5 register at the time you call the function and then restores the A5 value before calling your completion routine.

ASSEMBLY-LANGUAGE INFORMATION

The A0 register contains a pointer to the parameter block. You can look for the result code either in the `ioResult` field of the parameter block or in the D0 register.

MyRecipientStreamer

Your recipient stream function provides a method for processing data from the `OCEStreamRecipient` function.

```
pascal OSErr MyRecipientStreamer(void* buffer,
                              unsigned long count, Boolean eof,
                              long userData);
```

`buffer`	A pointer to the data that your stream method processes. This is supplied by the `OCEStreamRecipient` function each time it calls your recipient stream function.
`count`	The length, in bytes, of the current data in the buffer.
`eof`	A flag that the `OCEStreamRecipient` function sets when it last calls your recipient stream function. This flag signals that the `OCEStreamRecipient` function has finished processing the `OCERecipient` structure.
`userData`	The data that you supply in the `userData` parameter to the `OCEStreamRecipient` function. This data is passed directly to your recipient stream function.

DESCRIPTION

The `OCEStreamRecipient` function (page 7-111) calls your recipient stream function to process the data from an `OCERecipient` structure in discrete segments. You write this routine to process the data in the way that you want. The `OCEStreamRecipient` function calls your recipient stream function various times and passes your function progress information as well as the current portion of the `OCERecipient` to process. Any errors returned by this function are passed to the `OCEStreamRecipient` function.

SEE ALSO

The `OCERecipient` data structure is defined on page 7-24.

The `OCEStreamRecipient` function is described on page 7-111.

Summary of the IPM Manager

C Summary

Constants and Data Types

```
/* values of IPMPriority */
enum {
   kIPMAnyPriority = 0,
   kIPMNormalPriority = 1,
   kIPMLowPriority,
   kIPMHighPriority
};

typedef Byte IPMPriority;

/* values of IPMAccessMode */
enum {
   kIPMAtMark,
   kIPMFromStart,
   kIPMFromLEOM,
   kIPMFromMark
};

typedef unsigned short IPMAccessMode;

enum {
   kIPMUpdateMsgBit = 4,
   kIPMNewMsgBit = 5,
   kIPMDeleteMsgBit = 6
};

/* values of IPMNotificationType */
enum {
   kIPMUpdateMsgMask =  1<<kIPMUpdateMsgBit,
   kIPMNewMsgMask =     1<<kIPMNewMsgBit,
   kIPMDeleteMsgMask =  1<<kIPMDeleteMsgBit
};

typedef Byte IPMNotificationType;

/* values of IPMSenderTag */
enum {
   kIPMSenderRStringTag,
   kIPMSenderRecordIDTag
};

typedef unsigned short IPMSenderTag;

enum {
   kIPMFromDistListBit = 0,
   kIPMDummyRecBit = 1,
   kIPMFeedbackRecBit = 2,
   kIPMReporterRecBit = 3,
   kIPMBCCRecBit = 4
};

/* values of OCERecipientOffsetFlags */
enum {
   kIPMFromDistListMask =  1<<kIPMFromDistListBit,
   kIPMDummyRecMask =      1<<kIPMDummyRecBit,
   kIPMFeedbackRecMask =   1<<kIPMFeedbackRecBit,
   kIPMReporterRecMask =   1<<kIPMReporterRecBit,
   kIPMBCCRecMask =        1<<kIPMBCCRecBit
};

typedef Byte OCERecipientOffsetFlags;

#define kIPMTypeWildCard         'ipmw'

#define kIPMFamilyUnspecified    0
#define kIPMFamilyWildCard       0x3F3F3F3FL    /* '????' */

/* well-known signature */
#define kIPMSignature            'ipms'   /* base type */

/* well-known message types */
#define kIPMReportNotify         'rptn'   /* routing feedback */

/* well-known message block types */
```

```
#define kIPMEnclosedMsgType      'emsg'   /* enclosed (nested) message */
#define kIPMReportInfo           'rpti'   /* recipient information */
#define kIPMDigitalSignature     'dsig'   /* digital signature */

/* values of IPMMsgFormat */
enum {
   kIPMOSFormatType = 1,
   kIPMStringFormatType = 2
};

typedef unsigned short IPMMsgFormat;

/*
Following are the known extension values for IPM addresses handled by Apple
Computer, Inc.
*/

enum {
   kOCEalanXtn= 'alan',
   kOCEentnXtn= 'entn', /* 'entn' = entity name (aka DSSpec) */
   kOCEaphnXtn= 'aphn'
};

/* 'entn' extension forms */
enum {
   kOCEAddrXtn= 'addr', /* reserved */
   kOCEQnamXtn= 'qnam', /* queue-name form */
   kOCEAttrXtn= 'attr', /* an attribute specification */
   kOCESpAtXtn= 'spat'  /* specific attribute */
};

/* phoneNumber subtype constants */
enum {
   kOCEUseHandyDial = 1,
   kOCEDontUseHandyDial = 2
};

/* addresses with kIPMNBPXtn should specify this nbp type */
#define kIPMWSReceiverNBPType "\pMsgReceiver"

/* values of IPMHeaderSelector */
enum {
   kIPMTOC = 0,
   kIPMSender = 1,
   kIPMProcessHint = 2,
   kIPMMessageTitle = 3,
   kIPMMessageType = 4,
   kIPMFixedInfo = 7
};

typedef Byte IPMHeaderSelector;

enum {
   kIPMDeliveryNotificationBit      = 0,
   kIPMNonDeliveryNotificationBit   = 1,
   kIPMEncloseOriginalBit           = 2,
   kIPMSummaryReportBit             = 3,
   kIPMOriginalOnlyOnErrorBit       = 4
};

typedef Byte IPMNotificationType;

enum {
   kIPMNoNotificationMask           = 0x00,
   kIPMDeliveryNotificationMask     = 1<<kIPMDeliveryNotificationBit,
   kIPMNonDeliveryNotificationMask  = 1<<kIPMNonDeliveryNotificationBit,
   kIPMDontEncloseOriginalMask      = 0x00,
   kIPMEncloseOriginalMask          = 1<<kIPMEncloseOriginalBit,
   kIPMImmediateReportMask          = 0x00,
   kIPMSummaryReportMask            = 1<<kIPMSummaryReportBit,
   kIPMOriginalOnlyOnErrorMask      = 1<<kIPMOriginalOnlyOnErrorBit,
   kIPMEncloseOriginalOnErrorMask   =
                        (kIPMOriginalOnlyOnErrorMask|kIPMEncloseOriginalMask)
};

/* standard nondelivery codes */
enum {
   kIPMNoSuchRecipient              =  0x0001,
   kIPMRecipientMalformed           =  0x0002,
   kIPMRecipientAmbiguous           =  0x0003,
   kIPMRecipientAccessDenied        =  0x0004,
   kIPMGroupExpansionProblem        =  0x0005,
   kIPMMsgUnreadable                =  0x0006,
   kIPMMsgExpired                   =  0x0007,
   kIPMMsgNoTranslatableContent     =  0x0008,
   kIPMRecipientReqStdCont          =  0x0009,
   kIPMRecipientReqSnapShot         =  0x000A,
   kIPMNoTransferDiskFull           =  0x000B,
   kIPMNoTransferMsgRejectedbyDest  =  0x000C,
   kIPMNoTransferMsgTooLarge        =  0x000D
};

typedef unsigned long   IPMContextRef;

typedef unsigned long   IPMQueueRef;

typedef unsigned long   IPMMsgRef;

typedef unsigned long   IPMSeqNum;

typedef Str32 IPMProcHint;

typedef Str32 IPMQueueName;

typedef OCECreatorType IPMBlockType;
```

Message Addressing Structures

```
typedef DSSpec OCERecipient;

/* format of a packed form recipient */
#define OCEPackedRecipientHeader\
   unsigned short       dataLength;

struct ProtoOCEPackedRecipient {
   OCEPackedRecipientHeader
};
typedef struct ProtoOCEPackedRecipient ProtoOCEPackedRecipient;

# define kOCEPackedRecipientMAXBYTES (4096 - sizeof(ProtoOCEPackedRecipient))

struct OCEPackedRecipient {
   OCEPackedRecipientHeader
   Byte     data[kOCEPackedRecipientMaxBytes];
};
typedef struct OCEPackedRecipient OCEPackedRecipient;

struct IPMEntnQueueExtension {
   Str32 queueName;
};
typedef struct IPMEntnQueueExtension IPMEntnQueueExtension;
```

```
struct IPMEntnAttributeExtension {/* kOCEAttrXtn */
   AttributeType attributeName;
};
typedef struct IPMEntnAttributeExtension IPMEntnAttributeExtension;

struct IPMEntnSpecificAttributeExtension {   /* reserved */
   AttributeCreationID attributeCreationID;
   AttributeType attributeName;
};
typedef struct IPMEntnSpecificAttributeExtension
IPMEntnSpecificAttributeExtension;

struct IPMEntityNameExtension {
   OSType subExtensionType;
   union {
      IPMEntnSpecificAttributeExtension specificAttribute;
      IPMEntnAttributeExtension     attribute;
      IPMEntnQueueExtension         queue;
   } u;
};
typedef struct IPMEntityNameExtension IPMEntityNameExtension;
```

Message and Block Types

```
struct OCECreatorType {
   OSType         msgCreator;
   OSType         msgType;
};
typedef struct OCECreatorType OCECreatorType;

typedef Str32 IPMStringMsgType;

struct IPMMsgType {
   IPMMsgFormat         format;/* IPMMsgFormat*/
   union{
      OCECreatorType    msgOSType;
      IPMStringMsgType  msgStrType;
   }theType;
};
typedef struct IPMMsgType IPMMsgType;
```

Delivery Notification

```
struct IPMMsgID {
   unsigned long id[4];
   };
typedef struct IPMMsgID IPMMsgID;

struct IPMReportBlockHeader {
   IPMMsgID       msgID;         /* message ID of the original */
   UTCTime        creationTime;  /* creation time of the report */
};
typedef struct IPMReportBlockHeader IPMReportBlockHeader;

struct OCERecipientReport {
   unsigned short rcptIndex;     /* index of recipient in original message */
   OSErr          result;        /* result of sending letter to recipient */
};
typedef struct OCERecipientReport OCERecipientReport;
```

Filter Structures

```
struct IPMSingleFilter {   /* each field should be packed and word aligned */
   IPMPriority    priority;
   Byte           padByte;
   OSType         family;  /* family of this msg, '????' for all */
   ScriptCode     script;  /* language identifier */
   IPMProcHint    hint;
   IPMMsgType     msgType;
};
typedef struct IPMSingleFilter IPMSingleFilter;

struct IPMFilter {
   unsigned short    count;
   IPMSingleFilter   sFilters[1];
};
typedef struct IPMFilter IPMFilter;
```

Message Information Structure

```
struct IPMMsgInfo {  /* master message info */
   IPMSeqNum         sequenceNum;

   unsigned long     userData;
   unsigned short    respIndex;
   Byte              padByte;
   IPMPriority       priority;
   unsigned long     msgSize;
   unsigned short    originalRcptCount;
   unsigned short    reserved;
   UTCTime           creationTime;
   IPMMsgID          msgID;
   OSType            family;     /* family of this msg (e.g., mail) */
   IPMProcHint       procHint;   /* packed and even-length padded */
   IPMMsgType        msgType;    /* packed and even-length padded */
};
typedef struct IPMMsgInfo IPMMsgInfo;
```

Header Information Structures

```
struct IPMTOC {
   IPMBlockType      blockType;
   long              blockOffset;
   unsigned long     blockSize;
   unsigned long     blockRefCon;
};
typedef struct IPMTOC IPMTOC;

struct IPMFixedHdrInfo {
   unsigned short       version;          /* IPM Manager version */
   Boolean              authenticated;    /* was message authenticated? */
   Boolean              signatureEnclosed;/* digital signature enclosed? */
   unsigned long        msgSize;          /* size of message */
   IPMNotificationType  notification;     /* notification type requested */
   IPMPriority          priority;         /* message priority */
   unsigned short       blockCount;       /* number of blocks */
   unsigned short       originalRcptCount;/* original number of recipients */
   unsigned long        refCon;           /* application-defined data */
   unsigned short       reserved;         /* reserved */
   UTCTime              creationTime;     /* message creation time */
   IPMMsgID             msgID;            /* message ID */
   OSType               family;           /* family of this msg */
};
```

Sender Structure

```
struct IPMSender {
   IPMSenderTag      sendTag;
   union{
      RString        rString;
      PackedRecordID rid;
   } theSender;
};
typedef struct IPMSender IPMSender;
```

Parameter Block Header

```
#define IPMParamHeader     \
   Ptr      qLink;         \
   long     reservedH1;    \
   long     reservedH2;    \
   ProcPtr  ioCompletion;  \
   OSErr    ioResult;      \
   long     saveA5;        \
   short    reqCode;
```

Parameter Blocks for Creating a New Message

```
struct IPMNewMsgPB {
   IPMParamHeader
   unsigned long     filler;
   OCERecipient*     recipient;
   OCERecipient*     replyQueue;
   StringPtr         procHint;
   unsigned short    filler2;
   IPMMsgType*       msgType;
   unsigned long     refCon;
   IPMMsgRef         newMsgRef;
   unsigned short    filler3;
   long              filler4;
   AuthIdentity      identity;
   IPMSender*        sender;
   unsigned long     internalUse;
   unsigned long     internalUse2;
};
typedef struct IPMNewMsgPB IPMNewMsgPB;

struct IPMNewHFSMsgPB {
   IPMParamHeader
   FSSpec*           hfsPath;
   OCERecipient*     recipient;
   OCERecipient*     replyQueue;
   StringPtr         procHint;
   unsigned short    filler2;
   IPMMsgType*       msgType;
   unsigned long     refCon;
   IPMMsgRef         newMsgRef;
   unsigned short    filler3;
   long              filler4;
   AuthIdentity      identity;
   IPMSender*        sender;
   unsigned long     internalUse;
   unsigned long     internalUse2;
};
typedef struct IPMNewHFSMsgPB IPMNewHFSMsgPB;

typedef struct IPMAddRecipientPB {
   IPMParamHeader
   IPMMsgRef         msgRef;
   OCERecipient*     recipient;
   long              reserved;
};
typedef struct IPMAddRecipientPB IPMAddRecipientPB;

struct IPMAddReplyQueuePB {
   IPMParamHeader
   IPMMsgRef         msgRef;
   long              filler;
   OCERecipient*     replyQueue;
};
typedef struct IPMAddReplyQueuePB IPMAddReplyQueuePB;

struct IPMNewBlockPB {
   IPMParamHeader
   IPMMsgRef         msgRef;
   IPMBlockType      blockType;
   unsigned short    filler[5];
   unsigned long     refCon;
   unsigned short    filler2[3];
   long              startingOffset;
};
typedef struct IPMNewBlockPB IPMNewBlockPB;

struct IPMNewNestedMsgBlockPB {
   IPMParamHeader
   IPMMsgRef         msgRef;
   OCERecipient*     recipient;
   OCERecipient*     replyQueue;
   StringPtr         procHint;
   unsigned short    filler1;
   IPMMsgType*       msgType;
   unsigned long     refCon;
   IPMMsgRef         newMsgRef;
   unsigned short    filler2;
   long              startingOffset;
   AuthIdentity      identity;
   IPMSender*        sender;
   unsigned long     internalUse;
   unsigned long     internalUse2;
};
typedef struct IPMNewNestedMsgBlockPB IPMNewNestedMsgBlockPB;

struct IPMNestMsgPB {
   IPMParamHeader
   IPMMsgRef         msgRef;
   unsigned short    filler[9];
   unsigned long     refCon;
   IPMMsgRef         msgToNest;
   unsigned short    filler2;
   long              startingOffset;
};
typedef struct IPMNestMsgPB IPMNestMsgPB;

struct IPMWriteMsgPB {
   IPMParamHeader
   IPMMsgRef            msgRef;
   IPMAccessMode        mode;
   long                 offset;
   unsigned long        count;
   Ptr                  buffer;
   unsigned long        actualCount;
   Boolean              currentBlock;
};
typedef struct IPMWriteMsgPB IPMWriteMsgPB;

struct IPMEndMsgPB {
   IPMParamHeader
   IPMMsgRef            msgRef;
   IPMMsgID             msgID;
   RString*             msgTitle;
   IPMNotificationType  deliveryNotification;
   IPMPriority          priority;
   Boolean              cancel;
   Byte                 padByte;
   long                 reserved;
   SIGSignaturePtr      signature;
   Size                 signatureSize;
   SIGContextPtr        signatureContext;
   OSType               family;  /* family of this msg
                                    use kIPMFamilyUnspecified by default */
};
typedef struct IPMEndMsgPB IPMEndMsgPB;
```

Parameter Blocks for Managing Message Queues

```
struct IPMCreateQueuePB {
   IPMParamHeader
   long              filler1;
   OCERecipient*     queue;
   AuthIdentity      identity;/* used only if queue is remote */
   PackedRecordID*   owner;   /* used only if queue is remote */
};
typedef struct IPMCreateQueuePB IPMCreateQueuePB;

struct IPMOpenContextPB {
   IPMParamHeader
   IPMContextRef        contextRef;
};
typedef struct IPMOpenContextPB IPMOpenContextPB;

struct IPMOpenQueuePB {
   IPMParamHeader
   IPMContextRef        contextRef;
   OCERecipient*        queue;
   AuthIdentity         identity;
   IPMFilter*           filter;
   IPMQueueRef          newQueueRef;
   IPMNoteProcPtr       notificationProc;    /* must be nil */
   unsigned long        userData;            /* reserved */
   IPMNotificationType  noteType;            /* reserved */
   Byte                 padByte;             /* reserved */
   long                 reserved;
   long                 reserved2;
};
typedef struct IPMOpenQueuePB IPMOpenQueuePB;
typedef IPMEnumerateQueuePB   IPMChangeQueueFilterPB;

typedef IPMOpenContextPB      IPMCloseContextPB;

struct IPMCloseQueuePB {
   IPMParamHeader
   IPMQueueRef    queueRef;
};
typedef struct IPMCloseQueuePB IPMCloseQueuePB;

typedef IPMCreateQueuePB   IPMDeleteQueuePB;
```

Parameter Blocks for Listing and Reading Messages

```
struct IPMEnumerateQueuePB {
   IPMParamHeader
   IPMQueueRef    queueRef;
   IPMSeqNum      startSeqNum;
   Boolean        getProcHint;
   Boolean        getMsgType;
   short          filler;
   IPMFilter*     filter;
   unsigned short numToGet;
   unsigned short numGotten;
   unsigned long  enumCount;
   Ptr            enumBuffer;    /* will be packed array of IPMMsgInfo */
   unsigned long  actEnumCount;
};
typedef struct IPMEnumerateQueuePB IPMEnumerateQueuePB;

struct IPMOpenMsgPB {
   IPMParamHeader
   IPMQueueRef    queueRef;
   IPMSeqNum      sequenceNum;
   IPMMsgRef      newMsgRef;
   IPMSeqNum      actualSeqNum;
   Boolean        exactMatch;
   Byte           padByte;
   long           reserved;
};
typedef struct IPMOpenMsgPB IPMOpenMsgPB;

struct IPMOpenHFSMsgPB {
   IPMParamHeader
   FSSpec*        hfsPath;
   long           filler;
   IPMMsgRef      newMsgRef;
   long           filler2;
   Byte           filler3;
   long           reserved;
};
typedef struct IPMOpenHFSMsgPB IPMOpenHFSMsgPB;

struct IPMOpenBlockAsMsgPB {
   IPMParamHeader
   IPMMsgRef      msgRef;
   unsigned long  filler;
   IPMMsgRef      newMsgRef;
   unsigned short filler2[7];
   unsigned short blockIndex;
};
typedef struct IPMOpenBlockAsMsgPB IPMOpenBlockAsMsgPB;

struct IPMGetMsgInfoPB {
   IPMParamHeader
   IPMMsgRef               msgRef;
   IPMMsgInfo*             info;
};
typedef struct IPMGetMsgInfoPB IPMGetMsgInfoPB;

struct IPMReadHeaderPB {
   IPMParamHeader
   IPMMsgRef               msgRef;
   unsigned short          fieldSelector;
   long                    offset;
   unsigned long           count;
   Ptr                     buffer;
   unsigned long           actualCount;
   unsigned short          filler;
   unsigned long           remaining;
};
typedef struct IPMReadHeaderPB IPMReadHeaderPB;

struct IPMReadRecipientPB {
   IPMParamHeader
   IPMMsgRef               msgRef;
   unsigned short          rcptIndex;
   long                    offset;
   unsigned long           count;
   Ptr                      buffer;
   unsigned long            actualCount;
   short                    reserved;   /* must be 0 */
   unsigned long            remaining;
   unsigned short           originalIndex;
   OCERecipientOffsetFlags recipientOffsetFlags;
};
typedef struct IPMReadRecipientPB IPMReadRecipientPB;

typedef IPMReadRecipientPB IPMReadReplyQueuePB;

struct IPMGetBlkIndexPB {
   IPMParamHeader
   IPMMsgRef      msgRef;
   IPMBlockType   blockType;
   unsigned short index;
   unsigned short startingFrom;
   IPMBlockType   actualBlockType;
   unsigned short actualBlockIndex;
};
typedef struct IPMGetBlkIndexPB IPMGetBlkIndexPB;

struct IPMReadMsgPB {
   IPMParamHeader
   IPMMsgRef      msgRef;
   IPMAccessMode  mode;
   long           offset;
   unsigned long  count;
   Ptr            buffer;
   unsigned long  actualCount;
   unsigned short blockIndex;
   unsigned long  remaining;
};
typedef struct IPMReadMsgPB IPMReadMsgPB;

struct IPMVerifySignaturePB {
   IPMParamHeader
   IPMMsgRef      msgRef;
   SIGContextPtr  signatureContext;
};
typedef struct IPMVerifySignaturePB IPMVerifySignaturePB;
```

```
struct IPMCloseMsgPB {
   IPMParamHeader
   IPMMsgRef      msgRef;
   Boolean        deleteMsg;
};
typedef struct IPMCloseMsgPB IPMCloseMsgPB;
```

Parameter Block for Deleting Messages

```
struct IPMDeleteMsgRangePB {
   IPMParamHeader
   IPMQueueRef    queueRef;
   IPMSeqNum      startSeqNum;
   IPMSeqNum      endSeqNum;
   IPMSeqNum      lastSeqNum;
};
typedef struct IPMDeleteMsgRangePB IPMDeleteMsgRangePB;
```

Parameter Block Union Structure

```
union IPMParamBlock {
   struct {IPMParamHeader} header;
   IPMOpenContextPB        openContextPB;
   IPMCloseContextPB       closeContextPB;
   IPMCreateQueuePB        createQueuePB;
   IPMDeleteQueuePB        deleteQueuePB;
   IPMOpenQueuePB          openQueuePB;
   IPMCloseQueuePB         closeQueuePB;
   IPMEnumerateQueuePB     enumerateQueuePB;
   IPMChangeQueueFilterPB  changeQueueFilterPB;
   IPMDeleteMsgRangePB     deleteMsgRangePB;
   IPMOpenMsgPB            openMsgPB;
   IPMOpenHFSMsgPB         openHFSMsgPB;
   IPMOpenBlockAsMsgPB     openBlockAsMsgPB;
   IPMCloseMsgPB           closeMsgPB;
   IPMGetMsgInfoPB         getMsgInfoPB;
   IPMReadHeaderPB         readHeaderPB;
   IPMReadRecipientPB      readRecipientPB;
   IPMReadReplyQueuePB     readReplyQueuePB;
   IPMGetBlkIndexPB        getBlkIndexPB;
   IPMReadMsgPB            readMsgPB;
   IPMVerifySignaturePB    verifySignaturePB;
   IPMNewMsgPB             newMsgPB;
   IPMNewHFSMsgPB          newHFSMsgPB;
   IPMNestMsgPB            nestMsgPB;
   IPMNewNestedMsgBlockPB  newNestedMsgBlockPB;
   IPMEndMsgPB             endMsgPB;
   IPMAddRecipientPB       addRecipientPB;
   IPMAddReplyQueuePB      addReplyQueuePB;
   IPMNewBlockPB           newBlockPB;
   IPMWriteMsgPB           writeMsgPB;
};
typedef union IPMParamBlock IPMParamBlock;
typedef IPMParamBlock *IPMParamBlockPtr;
```

IPM Manager Functions

Creating a New Message

```
pascal OSErr IPMNewMsg      (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMNewHFSMsg   (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMAddRecipient
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMAddReplyQueue
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMNewBlock    (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMNewNestedMsgBlock
                            (IPMParamBlockPtr paramBlock,
                             Boolean async);
pascal OSErr IPMNestMsg     (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMWriteMsg    (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMEndMsg      (IPMParamBlockPtr paramBlock, Boolean async);
```

Managing Message Queues

```
pascal OSErr IPMCreateQueue
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMOpenContext
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMOpenQueue   (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMChangeQueueFilter
                            (IPMParamBlockPtr paramBlock,
                             Boolean async);
pascal OSErr IPMCloseQueue  (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMCloseContext
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMDeleteQueue
                            (IPMParamBlockPtr paramBlock, Boolean async);
```

Listing and Reading Messages

```
pascal OSErr IPMEnumerateQueue
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMOpenMsg     (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMOpenHFSMsg  (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMOpenBlockAsMsg
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMGetMsgInfo  (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMReadHeader
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMReadRecipient
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMReadReplyQueue
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMGetBlkIndex
                            (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMReadMsg     (IPMParamBlockPtr paramBlock, Boolean async);
pascal OSErr IPMVerifySignature
                            (IPMParamBlockPtr paramBlock);
pascal OSErr IPMCloseMsg    (IPMParamBlockPtr paramBlock, Boolean async);
```

Deleting Messages

```
pascal OSErr IPMDeleteMsgRange
                            (IPMParamBlockPtr paramBlock, Boolean async);
```

Utility Functions

```
pascal unsigned short OCESizePackedRecipient
                            (const OCERecipient *rcpt);
pascal unsigned short OCEPackRecipient
                            (const OCERecipient *rcpt, void* buffer);
pascal OSErr OCEUnpackRecipient
                            (const void* buffer, OCERecipient *rcpt,
                             RecordID *entitySpecifier);
```

```
pascal OSErr OCEStreamRecipient
                            (const OCERecipient* rcpt, OCERecipientStreamer
                             stream, long userData, unsigned long*
                             actualCount);
pascal OSType OCEGetRecipientType
                            (const CreationID *cid);
pascal void OCESetRecipientType
                            (OSType extensionType, CreationID *cid);
```

Application-Defined Functions

```
pascal void MyCompletionRoutine
                            (Ptr paramBlk);
pascal OSErr MyRecipientStreamer
                            (void* buffer, unsigned long count,
                             Boolean eof, long userData);
```

Pascal Summary

Constants

```
CONST

{ values of IPMPriority }
   kIPMAnyPriority                 = 0;     { for filter only}
   kIPMNormalPriority              = 1;

{ values of IPMAccessMode }
   kIPMAtMark                      = 0;
   kIPMFromStart                   = 1;
   kIPMFromLEOM                    = 2;
   kIPMFromMark                    = 3;

   kIPMUpdateMsgBit                = 4;
   kIPMNewMsgBit                   = 5;
   kIPMDeleteMsgBit                = 6;

{ values of IPMNotificationType }
   kIPMUpdateMsgMask               = $10;   {1<<kIPMUpdateMsgBit}
   kIPMNewMsgMask                  = $20;   {1<<kIPMNewMsgBit}
   kIPMDeleteMsgMask               = $40;   {1<<kIPMDeleteMsgBit}
```

```
{ values of IPMSenderTag }
   kIPMSenderRStringTag            = 0;
   kIPMSenderRecordIDTag           = 1;

   kIPMFromDistListBit             = 0;
   kIPMDummyRecBit                 = 1;
   kIPMFeedbackRecBit              = 2;     { redirect to feedback queue }
   kIPMReporterRecBit              = 3      { redirect to reporter original
                                             queue }

   kIPMBCCRecBit                   = 4;     { this recipient is blind to all
                                             recipients of message }

{ values of OCERecipientOffsetFlags }
   kIPMFromDistListMask            = $01;   {1<<kIPMFromDistListBit}
   kIPMDummyRecMask                = $02;   {1<<kIPMDummyRecBit}
   kIPMFeedbackRecMask             = $04;   {1<<kIPMFeedbackRecBit}
   kIPMReporterRecMask             = $08;   {1<<kIPMReporterRecBit}
   kIPMBCCRecMask                  = $10;   {1<<kIPMBCCRecBit}

   kIPMTypeWildCard                = 'ipmw';

   kIPMFamilyUnspecified           = 0;
   kIPMFamilyWildCard              = '????';

{ well known signature }
   kIPMSignature                   = 'ipms';{ base type }

{ well known message types }

   kIPMReportNotify                = 'rptn';{ routing feedback }

{ well known message block types }
   kIPMEnclosedMsgType             = 'emsg';{ enclosed (nested) message }
   kIPMReportInfo                  = 'rpti';{ recipient information }
   kIPMDigitalSignature            = 'dsig';{ digital signature }

{ values of IPMMsgFormat }
   kIPMOSFormatType                = 1;
   kIPMStringFormatType            = 2;
```

```
{Following are the known extension values for IPM addresses handled by Apple
Computer, Inc.}

   kOCEalanXtn                      = 'alan';
   kOCEentnXtn                      = 'entn';{ 'entn' = entity name
                                              (DSSpec: aka) }
   kOCEaphnXtn                      = 'aphn';

{ 'entn' extension forms }
   kOCEAddrXtn                      = 'addr';{ reserved }
   kOCEQnamXtn                      = 'qnam';{queue-name form }
   kOCEAttrXtn                      = 'attr';{ an attribute specification }
   kOCESpAtXtn                      = 'spat';{ specific attribute }

{ phoneNumber subtype constants }
   kOCEUseHandyDial                 = 1;
   kOCEDontUseHandyDial             = 2;

   kOCEPackedRecipientMaxBytes      =
                        (4096 - sizeof(ProtoOCEPackedRecipient));

{ addresses with kIPMNBPXtn should specify this nbp type }
   kIPMWSReceiverNBPType            = 'MsgReceiver';

{ values of IPMHeaderSelector }
   kIPMTOC                          = 0;
   kIPMSender                       = 1;
   kIPMProcessHint                  = 2;
   kIPMMessageTitle                 = 3;
   kIPMMessageType                  = 4;
   kIPMFixedInfo                    = 7;

   kIPMDeliveryNotificationBit      = 0;
   kIPMNonDeliveryNotificationBit   = 1;
   kIPMEncloseOriginalBit           = 2;
   kIPMSummaryReportBit             = 3;
   kIPMOriginalOnlyOnErrorBit       = 4;

   kIPMNoNotificationMask           = $00;
   kIPMDeliveryNotificationMask     = $01;  {1<<kIPMDeliveryNotificationBit}
   kIPMNonDeliveryNotificationMask  = $02;
                        {1<<kIPMNonDeliveryNotificationBit}
   kIPMDontEncloseOriginalMask      = $00;
   kIPMEncloseOriginalMask          = $04;  {1<<kIPMEncloseOriginalBit}
   kIPMImmediateReportMask          = $00;
   kIPMSummaryReportMask            = $08;  {1<<kIPMSummaryReportBit}
   kIPMOriginalOnlyOnErrorMask      = $10;  {1<<kIPMOriginalOnlyOnErrorBit}
   kIPMEncloseOriginalOnErrorMask   =
                     kIPMOriginalOnlyOnErrorMask + kIPMEncloseOriginalMask;

{ standard Nondelivery codes }
   kIPMNoSuchRecipient              = $0001;
   kIPMRecipientMalformed           = $0002;
   kIPMRecipientAmbiguous           = $0003;
   kIPMRecipientAccessDenied        = $0004;
   kIPMGroupExpansionProblem        = $0005;
   kIPMMsgUnreadable                = $0006;
   kIPMMsgExpired                   = $0007;
   kIPMMsgNoTranslatableContent     = $0008;
   kIPMRecipientReqStdCont          = $0009;
   kIPMRecipientReqSnapShot         = $000A;
   kIPMNoTransferDiskFull           = $000B;
   kIPMNoTransferMsgRejectedbyDest  = $000C;
   kIPMNoTransferMsgTooLarge        = $000D;
```

Data Types

```
TYPE

IPMPriority = Byte;

IPMAccessMode = INTEGER;

IPMNotificationType = Byte;

IPMSenderTag = INTEGER;

OCERecipientOffsetFlags = Byte;

IPMMsgFormat = INTEGER;

IPMHeaderSelector = Byte;

IPMContextRef = LONGINT;

IPMQueueRef = LONGINT;

IPMMsgRef = LONGINT;

IPMSeqNum = LONGINT;
```

```
IPMProcHint = Str32;

IPMQueueName = Str32;

IPMBlockType = OCECreatorType;
```

Message Addressing Structures

```
OCERecipient = DSSpec;

ProtoOCEPackedRecipient = RECORD
   dataLength:    INTEGER;
   END;

OCEPackedRecipient = RECORD
   dataLength:    INTEGER;
   data:          PACKED ARRAY[1..kOCEPackedRecipientMaxBytes] OF Byte;
   END;

OCEPackedRecipientPtr = ^OCEPackedRecipient;

IPMEntnQueueExtension = RECORD
   queueName: Str32;
   END;

IPMEntnAttributeExtension = RECORD{ kOCEAttrXtn }
   attributeName: AttributeType;
   END;

IPMEntnSpecificAttributeExtension = RECORD{ kOCESpAtXtn }
   attributeCreationID:    AttributeCreationID;
   attributeName:          AttributeType;
   END;

IPMEntityNameExtension = RECORD
   subExtensionType: OSType;
   CASE INTEGER OF
      1: (specificAttribute:  IPMEntnSpecificAttributeExtension);
      2: (attribute:          IPMEntnAttributeExtension);
      3: (queue:              IPMEntnQueueExtension);
   END;
```

Message and Block Types

```
OCECreatorType = RECORD
   msgCreator:       OSType;
   msgType:          OSType;
   END;

IPMStringMsgType = Str32;

IPMMsgType = RECORD
   format: IPMMsgFormat;{ IPMMsgFormat}
   CASE INTEGER OF
      1: (msgOSType:    OCECreatorType);
      2: (msgStrType:   IPMStringMsgType);
   END;
```

Delivery Notification Structures

```
IPMMsgID = RECORD
   id: ARRAY[1..4] OF LONGINT;
   END;

IPMReportBlockHeader = RECORD
   msgID:            IPMMsgID;{ message ID of the original }
   creationTime:     UTCTime;{ creation time of the report }
   END;

OCERecipientReport = RECORD
   rcptIndex:        INTEGER;{ index of recipient in original message }
   result:           OSErr;   { result of sending letter to this recipient}
   END;
```

Filter Structures

```
IPMSingleFilter = PACKED RECORD
   priority:   IPMPriority;
   padByte:    Byte;
   family:     OSType;     { family of this msg, '????' for all }
   script:     ScriptCode; { language identifier }
   hint:       IPMProcHint;
   msgType:    IPMMsgType;
   END;
```

```
IPMFilter = RECORD
   count:      INTEGER;
   sFilters:   ARRAY[1..1] OF IPMSingleFilter;
   END;
```

Message Information Structure

```
IPMMsgInfo = PACKED RECORD
   sequenceNum:        IPMSeqNum;
   userData:           LONGINT;
   respIndex:          INTEGER;
   padByte:            Byte;
   priority:           IPMPriority;
   msgSize:            LONGINT;
   originalRcptCount:  INTEGER;
   reserved:           INTEGER;
   creationTime:       UTCTime;
   msgID:              IPMMsgID;
   family:             OSType;        { family of this msg
                                        (e.g. mail) }
   procHint:           IPMProcHint;   { packed and even-length padded }
   msgType:            IPMMsgType;    { packed and even-length padded }
   END;
```

Header Information Structures

```
IPMTOC = RECORD
   blockType:        IPMBlockType;
   blockOffset:      LONGINT;
   blockSize:        LONGINT;
   blockRefCon:      LONGINT;
   END;

IPMFixedHdrInfo = PACKED RECORD
   version:            INTEGER;            { IPM Manager version }
   authenticated:      BOOLEAN;            { was message authenticated? }
   signatureEnclosed:  BOOLEAN;            { digital signature enclosed? }
   msgSize:            LONGINT;            { size of message }
   notification:       IPMNotificationType;{ notification type requested }
   priority:           IPMPriority;        { message priority }
   blockCount:         INTEGER;            { number of blocks }
   originalRcptCount:  INTEGER;            { original number of recipients }
   refCon:             LONGINT;            { application-defined data }
   reserved:           INTEGER;            { reserved }
   creationTime:        UTCTime;            { message creation time }
   msgID:               IPMMsgID;           { message ID }
   family:              OSType;             { family of this msg }
   END;
```

Sender Structure

```
IPMSender = RECORD
   sendTag: IPMSenderTag;
   CASE INTEGER OF
      1: (rString: RString);
      2: (rid:     PackedRecordID);
   END;
```

Parameter Block Header

```
IPMParamHeader = RECORD
   qLink:         Ptr;
   reservedH1:    LONGINT;
   reservedH2:    LONGINT;
   ioCompletion:  ProcPtr;
   ioResult:      OSErr;
   saveA5:        LONGINT;
   reqCode:       INTEGER;
   END;
```

Parameter Blocks for Creating a New Message

```
IPMNewMsgPB = RECORD
   qLink: Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:           LONGINT;
   reqCode:          INTEGER;
   filler:           LONGINT;
   recipient:        ^OCERecipient;
   replyQueue:       ^OCERecipient;
   procHint:         StringPtr;
   filler2:          INTEGER;
   msgType:          ^IPMMsgType;
   refCon:           LONGINT;
   newMsgRef:        IPMMsgRef;
   filler3:          INTEGER;
   filler4:          LONGINT;
   identity:         AuthIdentity;
   sender:           ^IPMSender;
   internalUse:      LONGINT;
   internalUse2:     LONGINT;
   END;

IPMNewHFSMsgPB = RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:            LONGINT;
   reqCode:          INTEGER;
   hfsPath:          ^FSSpec;
   recipient:        ^OCERecipient;
   replyQueue:       ^OCERecipient;
   procHint:         StringPtr;
   filler2:          INTEGER;
   msgType:          ^IPMMsgType;
   refCon:           LONGINT;
   newMsgRef:        IPMMsgRef;
   filler3:          INTEGER;
   filler4:          LONGINT;
   identity:         AuthIdentity;
   sender:           ^IPMSender;
   internalUse:      LONGINT;
   internalUse2:     LONGINT;
   END;

IPMAddRecipientPB = RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:           LONGINT;
   reqCode:          INTEGER;
   msgRef:           IPMMsgRef;
   recipient:        ^OCERecipient;
   reserved:         LONGINT;
   END;
```

```
IPMAddReplyQueuePB = RECORD
   qLink:             Ptr;
   reservedH1:        LONGINT;
   reservedH2:        LONGINT;
   ioCompletion:      ProcPtr;
   ioResult:          OSErr;
   saveA5:            LONGINT;
   reqCode:           INTEGER;
   msgRef:            IPMMsgRef;
   filler:            LONGINT;
   replyQueue:        ^OCERecipient;
   END;

IPMNewBlockPB = RECORD
   qLink:             Ptr;
   reservedH1:        LONGINT;
   reservedH2:        LONGINT;
   ioCompletion:      ProcPtr;
   ioResult:          OSErr;
   saveA5:            LONGINT;
   reqCode:           INTEGER;
   msgRef:            IPMMsgRef;
   blockType:         IPMBlockType;
   filler:            ARRAY[1..5] OF INTEGER;
   refCon:            LONGINT;
   filler2:           ARRAY[1..3] OF INTEGER;
   startingOffset:    LONGINT;
   END;

IPMNewNestedMsgBlockPB = RECORD
   qLink:             Ptr;
   reservedH1:        LONGINT;
   reservedH2:        LONGINT;
   ioCompletion:      ProcPtr;
   ioResult:          OSErr;
   saveA5:            LONGINT;
   reqCode:           INTEGER;
   msgRef:            IPMMsgRef;
   recipient:         ^OCERecipient;
   replyQueue:        ^OCERecipient;
   procHint:          StringPtr;
   filler1:           INTEGER;
   msgType:           ^IPMMsgType;
   refCon:            LONGINT;
   newMsgRef:          IPMMsgRef;
   filler2:            INTEGER;
   startingOffset:     LONGINT;
   identity:           AuthIdentity;
   sender:             ^IPMSender;
   internalUse:        LONGINT;
   internalUse2:       LONGINT;
   END;

IPMNestMsgPB = RECORD
   qLink:              Ptr;
   reservedH1:         LONGINT;
   reservedH2:         LONGINT;
   ioCompletion:       ProcPtr;
   ioResult:           OSErr;
   saveA5:             LONGINT;
   reqCode:            INTEGER;
   msgRef:             IPMMsgRef;
   filler:             ARRAY[1..9] OF INTEGER;
   refCon:             LONGINT;
   msgToNest:          IPMMsgRef;
   filler2:            INTEGER;
   startingOffset:     LONGINT;
   END;

IPMWriteMsgPB = RECORD
   qLink:              Ptr;
   reservedH1:         LONGINT;
   reservedH2:         LONGINT;
   ioCompletion:       ProcPtr;
   ioResult:           OSErr;
   saveA5:             LONGINT;
   reqCode:            INTEGER;
   msgRef:             IPMMsgRef;
   mode:               IPMAccessMode;
   offset:             LONGINT;
   count:              LONGINT;
   buffer:             Ptr;
   actualCount:        LONGINT;
   currentBlock:       BOOLEAN;
   END;
```

```
IPMEndMsgPB = PACKED RECORD
   qLink:               Ptr;
   reservedH1:          LONGINT;
   reservedH2:          LONGINT;
   ioCompletion:        ProcPtr;
   ioResult:            OSErr;
   saveA5:              LONGINT;
   reqCode:             INTEGER;
   msgRef:              IPMMsgRef;
   msgID:               IPMMsgID;
   msgTitle:            ^RString;
   deliveryNotification: IPMNotificationType;
   priority:            IPMPriority;
   cancel:              BOOLEAN;
   padByte:             Byte;
   reserved:            LONGINT;
   signature:           SIGSignaturePtr;
   signatureSize:       Size;
   signatureContext:    SIGContextPtr;
   family:              OSType;           { family of this msg (e.g.,
                                           mail) use kIPMFamilyUnspecified
                                           by default }
   END;
```

Parameter Blocks for Managing Message Queues

```
IPMCreateQueuePB = RECORD
   qLink:               Ptr;
   reservedH1:          LONGINT;
   reservedH2:          LONGINT;
   ioCompletion:        ProcPtr;
   ioResult:            OSErr;
   saveA5:              LONGINT;
   reqCode:             INTEGER;
   filler1:             LONGINT;
   queue:               ^OCERecipient;
   identity:            AuthIdentity;     { used only if queue is remote }
   owner:               ^PackedRecordID;  { used only if queue is remote }
   END;

IPMOpenContextPB = RECORD
   qLink:               Ptr;
   reservedH1:          LONGINT;
   reservedH2:          LONGINT;
   ioCompletion:        ProcPtr;
   ioResult:            OSErr;
   saveA5:              LONGINT;
   reqCode:             INTEGER;
   contextRef:          IPMContextRef; { context reference to be used in
                                          further calls}
   END;

IPMOpenQueuePB = PACKED RECORD
   qLink:               Ptr;
   reservedH1:          LONGINT;
   reservedH2:          LONGINT;
   ioCompletion:        ProcPtr;
   ioResult:            OSErr;
   saveA5:              LONGINT;
   reqCode:             INTEGER;
   contextRef:          IPMContextRef;
   queue:               ^OCERecipient;
   identity:            AuthIdentity;
   filter:              ^IPMFilter;
   newQueueRef:         IPMQueueRef;
   notificationProc:    IPMNoteProcPtr;
   userData:            LONGINT;
   noteType:            IPMNotificationType;
   padByte:             Byte;
   reserved:            LONGINT;
   reserved2:           LONGINT;
   END;

IPMChangeQueueFilterPB = IPMEnumerateQueuePB;

IPMCloseContextPB = IPMOpenContextPB;

IPMCloseQueuePB = RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:           LONGINT;
   reqCode:          INTEGER;
   queueRef:         IPMQueueRef;
   END;

IPMDeleteQueuePB = IPMCreateQueuePB;
```

Parameter Blocks for Listing and Reading Messages

```
IPMEnumerateQueuePB = RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:           LONGINT;
   reqCode:          INTEGER;
   queueRef:         IPMQueueRef;
   startSeqNum:      IPMSeqNum;
   getProcHint:      BOOLEAN;
   getMsgType:       BOOLEAN;
   filler:           INTEGER;
   filter:           ^IPMFilter;
   numToGet:         INTEGER;
   numGotten:        INTEGER;
   enumCount:        LONGINT;
   enumBuffer:       Ptr;        { will be packed array of IPMMsgInfo }
   actEnumCount:     LONGINT;
   END;

IPMOpenMsgPB = PACKED RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:           LONGINT;
   reqCode:          INTEGER;
   queueRef:         IPMQueueRef;
   sequenceNum:         IPMSeqNum;
   newMsgRef:           IPMMsgRef;
   actualSeqNum:        IPMSeqNum;
   exactMatch:          BOOLEAN;
   padByte:             Byte;
   reserved:            LONGINT;
   END;

IPMOpenHFSMsgPB = RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:           LONGINT;
   reqCode:          INTEGER;
   hfsPath:          ^FSSpec;
   filler:           LONGINT;
   newMsgRef:        IPMMsgRef;
   filler2:          LONGINT;
   filler3:          Byte;
   reserved:         LONGINT;
   END;

IPMOpenBlockAsMsgPB = RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:           LONGINT;
   reqCode:          INTEGER;
   msgRef:           IPMMsgRef;
   filler:           LONGINT;
   newMsgRef:        IPMMsgRef;
   filler2:          ARRAY[1..7] OF INTEGER;
   blockIndex:       INTEGER;
   END;

IPMGetMsgInfoPB = RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:         OSErr;
   saveA5:           LONGINT;
   reqCode:          INTEGER;
   msgRef:           IPMMsgRef;
   info:             ^IPMMsgInfo;
   END;

IPMReadHeaderPB = RECORD
   qLink:            Ptr;
   reservedH1:       LONGINT;
   reservedH2:       LONGINT;
   ioCompletion:     ProcPtr;
   ioResult:          OSErr;
   saveA5:            LONGINT;
   reqCode:           INTEGER;
   msgRef:            IPMMsgRef;
   fieldSelector:     INTEGER;
   offset:            LONGINT;
   count:             LONGINT;
   buffer:            Ptr;
   actualCount:       LONGINT;
   filler:            INTEGER;
   remaining:         LONGINT;
   END;

IPMReadRecipientPB = PACKED RECORD
   qLink:                Ptr;
   reservedH1:           LONGINT;
   reservedH2:           LONGINT;
   ioCompletion:         ProcPtr;
   ioResult:             OSErr;
   saveA5:               LONGINT;
   reqCode:              INTEGER;
   msgRef:               IPMMsgRef;
   rcptIndex:            INTEGER;
   offset:               LONGINT;
   count:                LONGINT;
   buffer:               Ptr;
   actualCount:          LONGINT;
   reserved:             INTEGER;    { must be 0 }
   remaining:            LONGINT;
   originalIndex:        INTEGER;
   recipientOffsetFlags: OCERecipientOffsetFlags;
   END;

IPMReadReplyQueuePB = IPMReadRecipientPB;

IPMGetBlkIndexPB = RECORD
   qLink:                Ptr;
   reservedH1:           LONGINT;
   reservedH2:           LONGINT;
   ioCompletion:         ProcPtr;
   ioResult:             OSErr;
   saveA5:               LONGINT;
   reqCode:              INTEGER;
   msgRef:               IPMMsgRef;
   blockType:             IPMBlockType;
   index:                 INTEGER;
   startingFrom:          INTEGER;
   actualBlockType:       IPMBlockType;
   actualBlockIndex:      INTEGER;
   END;

IPMReadMsgPB = RECORD
   qLink:                 Ptr;
   reservedH1:            LONGINT;
   reservedH2:            LONGINT;
   ioCompletion:          ProcPtr;
   ioResult:              OSErr;
   saveA5:                LONGINT;
   reqCode:               INTEGER;
   msgRef:                IPMMsgRef;
   mode:                  IPMAccessMode;
   offset:                LONGINT;
   count:                 LONGINT;
   buffer:                Ptr;
   actualCount:           LONGINT;
   blockIndex:            INTEGER;
   remaining:             LONGINT;
   END;

IPMVerifySignaturePB = RECORD
   qLink:                 Ptr;
   reservedH1:            LONGINT;
   reservedH2:            LONGINT;
   ioCompletion:          ProcPtr;
   ioResult:              OSErr;
   saveA5:                LONGINT;
   reqCode:               INTEGER;
   msgRef:                IPMMsgRef;
   signatureContext:      SIGContextPtr;
   END;

IPMCloseMsgPB = RECORD
   qLink:                 Ptr;
   reservedH1:            LONGINT;
   reservedH2:            LONGINT;
   ioCompletion:          ProcPtr;
   ioResult:              OSErr;
   saveA5:                LONGINT;
   reqCode:             INTEGER;
   msgRef:              IPMMsgRef;
   deleteMsg:           BOOLEAN;
   END;
```

Parameter Blocks for Deleting Messages

```
IPMDeleteMsgRangePB = RECORD
   qLink:               Ptr;
   reservedH1:          LONGINT;
   reservedH2:          LONGINT;
   ioCompletion:        ProcPtr;
   ioResult:            OSErr;
   saveA5:              LONGINT;
   reqCode:             INTEGER;
   queueRef:            IPMQueueRef;
   startSeqNum:         IPMSeqNum;
   endSeqNum:           IPMSeqNum;
   lastSeqNum:          IPMSeqNum;
   END;
```

Parameter Block Union Structure

```
IPMParamBlock = RECORD
   CASE INTEGER OF
       1:(header: IPMParamHeader);
       2:(openContextPB: IPMOpenContextPB);
       3:(closeContextPB: IPMCloseContextPB);
       4:(createQueuePB: IPMCreateQueuePB);
       5:(deleteQueuePB: IPMDeleteQueuePB);
       6:(openQueuePB: IPMOpenQueuePB);
       7:(closeQueuePB: IPMCloseQueuePB);
       8:(enumerateQueuePB: IPMEnumerateQueuePB);
       9:(changeQueueFilterPB: IPMChangeQueueFilterPB);
      10:(deleteMsgRangePB: IPMDeleteMsgRangePB);
      11:(openMsgPB: IPMOpenMsgPB);
      12:(openHFSMsgPB: IPMOpenHFSMsgPB);
      13:(openBlockAsMsgPB: IPMOpenBlockAsMsgPB);
      14:(closeMsgPB: IPMCloseMsgPB);
      15:(getMsgInfoPB: IPMGetMsgInfoPB);
      16:(readHeaderPB: IPMReadHeaderPB);
      17:(readRecipientPB: IPMReadRecipientPB);
      18:(readReplyQueuePB: IPMReadReplyQueuePB);
      19:(getBlkIndexPB: IPMGetBlkIndexPB);
      20:(readMsgPB: IPMReadMsgPB);
      21:(verifySignaturePB: IPMVerifySignaturePB);
      22:(newMsgPB: IPMNewMsgPB);
      23:(newHFSMsgPB: IPMNewHFSMsgPB);
      24:(nestMsgPB: IPMNestMsgPB);
      25:(newNestedMsgBlockPB: IPMNewNestedMsgBlockPB);
      26:(endMsgPB: IPMEndMsgPB);
      27:(addRecipientPB: IPMAddRecipientPB);
      28:(addReplyQueuePB: IPMAddReplyQueuePB);
      29:(newBlockPB: IPMNewBlockPB);
      30:(writeMsgPB: IPMWriteMsgPB);
   END;

IPMParamBlockPtr = ^IPMParamBlock;
```

IPM Manager Functions

Creating a New Message

```
FUNCTION IPMNewMsg            (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
FUNCTION IPMNewHFSMsg         (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
FUNCTION IPMAddRecipient      (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
FUNCTION IPMAddReplyQueue     (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
FUNCTION IPMNewBlock          (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
FUNCTION IPMNewNestedMsgBlock
                              (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
FUNCTION IPMNestMsg           (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
FUNCTION IPMWriteMsg          (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
FUNCTION IPMEndMsg            (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                               OSErr;
```

Managing Message Queues

```
FUNCTION IPMCreateQueue     (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMOpenContext     (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMOpenQueue       (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMChangeQueueFilter
                            (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMCloseQueue      (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMCloseContext    (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMDeleteQueue     (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
```

Listing and Reading Messages

```
FUNCTION IPMEnumerateQueue  (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMOpenMsg         (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMOpenHFSMsg      (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMOpenBlockAsMsg  (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMGetMsgInfo      (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMReadHeader      (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMReadRecipient   (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMReadReplyQueue  (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMGetBlkIndex     (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
FUNCTION IPMReadMsg         (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
```

```
FUNCTION IPMVerifySignature
                            (paramBlock: IPMParamBlockPtr): OSErr;{ Always
                             synchronous }
FUNCTION IPMCloseMsg        (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
```

Deleting Messages

```
FUNCTION IPMDeleteMsgRange  (paramBlock: IPMParamBlockPtr; async: BOOLEAN):
                             OSErr;
```

Utility Routines

```
FUNCTION OCESizePackedRecipient
                            (rcpt: OCERecipient): INTEGER;
FUNCTION OCEPackRecipient   (rcpt: OCERecipient; buffer: UNIV Ptr): INTEGER;
FUNCTION OCEUnpackRecipient
                            (buffer: UNIV Ptr; VAR rcpt: OCERecipient;
                             VAR entitySpecifier: RecordID): OSErr;
FUNCTION OCEStreamRecipient (rcpt: OCERecipient; stream:
                             OCERecipientStreamer;
                             userData: LONGINT; VAR actualCount: LONGINT):
                             OSErr;
FUNCTION OCEGetRecipientType
                            (cid: CreationID): OSType;
PROCEDURE OCESetRecipientType
                            (extensionType: OSType; VAR cid: CreationID);
```

Application-Defined Functions

```
FUNCTION MyCompletionRoutine
                            (paramBlk: Ptr);
FUNCTION MyRecipientStreamer
                            (VAR buffer: void; count: LONGINT; eof:
                             BOOLEAN; userData: LONGINT): OSErr;)
```

Assembly-Language Summary

Trap Macros Requiring Routine Selectors

`__OCETBDispatch`

Selector	Routine
$0400	`IPMOpenContext`
$0401	`IPMCloseContext`
$0402	`IPMNewMsg`
$0403	`IPMAddRecipient`
$0404	`IPMNewBlock`
$0405	`IPMNewNestedMsgBlock`
$0406	`IPMNestMsg`
$0407	`IPMWriteMsg`
$0408	`IPMEndMsg`
$0409	`IPMOpenQueue`
$040A	`IPMCloseQueue`
$040B	`IPMOpenMsg`
$040C	`IPMCloseMsg`
$040D	`IPMReadMsg`
$040E	`IPMReadHeader`
$040F	`IPMOpenBlockAsMsg`
$0410	`IPMReadRecipient`
$0411	`IPMCreateQueue`
$0412	`IPMDeleteQueue`
$0413	`IPMEnumerateQueue`
$0414	`IPMChangeQueueFilter`
$0415	`IPMDeleteMsgRange`
$0417	`IPMOpenHFSMsg`
$0418	`IPMGetBlkIndex`
$0419	`IPMGetMsgInfo`
$041D	`IPMAddReplyQueue`
$041E	`IPMNewHFSMsg`
$0421	`IPMReadReplyQueue`
$0422	`IPMVerifySignature`

`__OCEMessaging`

Selector	Routine
$033E	`OCESizePackedRecipient`
$033F	`OCEPackRecipient`
$0340	`OCEUnpackRecipient`
$0341	`OCEStreamRecipient`
$0342	`OCEGetRecipientType`
$0343	`OCESetRecipientType`

Result Codes

The allocated range of result codes for the Interprogram Messaging Manager is –15090 through –15169. Routines may also return result codes from other AOCE managers and standard Macintosh result codes such as `noErr` 0 (no error) and `fnfErr` –43 (file not found).

`noErr`	0	No error
`kOCEParamErr`	–50	Parameter error
`kOCEConnectionClosed`	–1513	Network connection has closed
`kIPMCantCreateIPMCatEntry`	–15090	Internal error
`kIPMInvalidMsgType`	–15091	Message type is invalid
`kIPMInvalidProcHint`	–15092	Process hint is invalid
`kIPMInvalidOffset`	–15093	Bad offset for read or write operation
`kIPMUpdateCatFailed`	–15094	Internal error
`kIPMMsgTypeReserved`	–15095	Message type reserved for system use
`kIPMNotInABlock`	–15096	Specified starting point not within the message
`kIPMNestedMsgOpened`	–15097	Nested message opened; cannot do operation
`kIPMA1HdrCorrupt`	–15098	Message is corrupt; may not be message
`kIPMCorruptDataStructures`	–15099	Message is corrupt
`kIPMAbortOfNestedMsg`	–15100	Canceled nested message
`kIPMBlockIsNotNestedMsg`	–15101	Block is not message (`IPMOpenBlockAsMsg`)
`kIPMCacheFillError`	–15102	Internal error
`kIPMInvalidSender`	–15103	Sender is invalid
`kIPMNoRecipientsYet`	–15104	Require recipient to send
`kIPMInvalidFilter`	–15105	Filter is invalid
`kIPMAttrNotInHdr`	–15106	Specified attribute not in message header
`kIPMBlkNotFound`	–15107	Specified block nonexistent
`kIPMStreamErr`	–15108	Error on stream
`kIPMPortClosed`	–15109	Stream closed
`kIPMBinBusy`	–15110	Internal error
`kIPMCorruptedBin`	–15111	IPM BIN is damaged
`kIPMBadQName`	–15112	Invalid queue name
`kIPMEndOfBin`	–15113	Internal error
`kIPMBinNeedsConversion`	–15114	IPM BIN needs conversion
`kIPMMgrInternalErr`	–15115	Internal error
`kIPMEltBusy`	–15116	Message or letter opened (on delete operation)
`kIPMEltClosedNotDeleted`	–15117	Element was closed but not deleted

`kIPMBadContext`	–15118	Invalid reference
`kIPMContextIsClosing`	–15119	Reference is closing
`kIPMeoQ`	–15120	No more messages (`IPMEnumerateQueue`)
`kIPMEltNotFound`	–15122	No such item or message
`kIPMQBusy`	–15126	Specified queue busy; cannot delete
`kIPMLookupAttrTooBig`	–15129	Attribute in lookup is too big
`kIPMAccessDenied`	–15141	Access denied
`kIPMNoAttrsFound`	–15146	No attributes found in lookup
`kIPMBadMailSlotAttrVal`	–15149	Invalid mail slot attribute value