Oracle® Communications Messaging Server

MTA Developer's Reference

Release 8.1.0

F15153-01

March 2019

ORACLE®

Oracle Communications Messaging Server MTA Developer's Reference, Release 8.1.0

F15153-01

Contents

3 Enqueuing Messages

4 Dequeuing Messages

5 Decoding Messages

6 MTA SDK Reference

7 Using Callable Send mtaSend()

8 mtaSend() Routine Specification

Preface

This guide explains how to use the Oracle Communications Messaging Server MTA SDK.

Audience

This document is intended for developers. This guide assumes you are familiar with the following topics:

- Messaging protocols
- System administration and networking
- General deployment architectures

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents in the Messaging Server documentation set:

- *Messaging Server Installation and Configuration Guide*: Provides instructions for installing and configuring Messaging Server.
- *Messaging Server Installation and Configuration Guide for Cassandra Message Store*: Provides instructions for installing and configuring Cassandra message store.
- *Messaging Server Reference*: Provides additional information for using and configuring Messaging Server.
- *Messaging Server Release Notes*: Describes the new features, fixes, known issues, troubleshooting tips, and required third-party products and licensing.

- *Messaging Server Security Guide*: Provides guidelines and recommendations for setting up Messaging Server in a secure configuration.
- *Messaging Server System Administrator's Guide*: Describes how to administer Messaging Server and its accompanying software components. This document is intended system administrators.

1

MTA SDK Concepts and Overview

This chapter provides an overview of the Oracle Communications Messaging Server MTA SDK, which is a low-level interface, with routines falling into three categories: those that enqueue messages, those that dequeue messages, and miscellaneous routines that typically query or set MTA states, or parse message structures, such as lists of RFC 822 addresses.

The Callable Send facility, described in "Decoding Messages" and "MTA SDK Reference" and used only for originating mail from the local host, can be used simultaneously with the MTA SDK.

Channel Programs and Message Queuing

Message enqueuing and dequeuing are generally done by channel programs also referred to simply as channels. There are two types of channel programs, master channel that dequeue messages, and channels (sometimes referred to as slave channels) that enqueue messages. Each MTA channel has its own message queue, referred to as a channel queue. Channel programs may also perform intermediate roles by dequeuing messages from one message queue and re-enqueuing them to another while, typically, processing the message at the same time. For example, the message processing might be to convert the message body from one format to another.

Managing Multiple Threads Using Contexts

A number of SDK operations require multiple, sequential calls to the SDK routines. To manage this, the SDK provides the caller with a pointer to an opaque data structure called a context. This mechanism allows for management of state information across calls to the SDK. Use of the contexts allows multiple threads within a single program to make simultaneous calls to the same SDK routine. The only limitation is that a single, specific context may not be simultaneously used by different threads in calls to the SDK. When such usage is detected in an SDK call, an **MTA_THREAD** error results.

In some cases these contexts are automatically created for you, such as dequeue and decode contexts. In all other cases, for example for enqueue contexts, you must make an explicit call to create them. The calls that automatically create contexts also automatically dispose of them. In all other cases, a call must be made to explicitly dispose of a context. It is important to dispose of contexts when you no longer need them as so doing releases resources such as virtual memory.

For more information on contexts, see "Threads and Enqueue Contexts" and "Threads and Dequeue Contexts".

Enqueuing Messages

Messages are introduced to the MTA by enqueuing them. Each enqueued message contains two required components, the message envelope and the message header, and may optionally contain a third component, the message body. The contents of the envelope and header must be provided by the program using the SDK.

For instructions about enqueuing messages, see "MTA SDK Programming Considerations".

For an example of how to enqueue a message, see "A Simple Example of Enqueuing a Message".

Message Components

This section describes the three message components: envelope, header and body.

Envelope

The message envelope contains the envelope From: address, and the list of envelope To: addresses. The envelope is created by the SDK as the message is enqueued. The addresses to be placed in the envelope must conform to RFC 2822. The envelope To: addresses are often referred to as envelope recipient addresses.

Programs should rely solely upon the MTA SDK routines to read and write envelope information, since the queued message file formats are subject to change. Using the SDK routines insulates programmers from format changes.

The routines mtaEnqueueStart() and mtaEnqueueTo() are used to construct a message envelope.

Header

The message header contains RFC 2822 style header lines. The program enqueuing the message has nearly complete control over the contents of the header and can specify as many or as few header lines as it sees fit, with a few exceptions. A header must have at a minimum three lines: **From:**, **Date:**, and at least one recipient header line, such as **To:**, **Cc:**, or **Bcc:**.

As the message is enqueued, the SDK will do its best to supply any mandatory header lines that are missing as well as take some measures to ensure that the contents of the header lines conform to any relevant standards. If the **From:** header line is omitted by the program using the SDK, the SDK code will construct a default header line from the envelope **From:** address. This may not always be appropriate; for instance, when mail is addressed to a mailing list that specifies an **Errors-to:** address, then the **Errors-to**: address should be used as the envelope **From:** address. In this case, it is not appropriate to derive the header **From:** line from the envelope **From:** address. If the **Date:** header line is omitted, the SDK code will supply it, as well as a **Date-warning:** header line. Finally, if no recipient header lines are present, then the SDK code will generate them using the envelope recipient addresses.

Any addresses appearing in the message header should conform to RFC 2822.

The header is written line-by-line using the routines mtaEnqueueWrite() and mtaEnqueueWriteLine().

Body

The optional message body contains the content of the message. As with the message header, the program enqueuing the message has nearly complete control over the contents of the message body. The only exception to this is when the message is structured with multiple parts or requires encoding, for example if it contains binary

data, or lines requiring wrapping. In such cases, the SDK will ensure that the message body conforms to MIME standards (RFCs 2045- 2049).

As with the message header, message body lines are written with the routines mtaEnqueueWrite() and mtaEnqueueWriteLine().

A Sample Enqueued Message

Enqueued messages may be seen in the MTA queue directories and are merely ASCII text files. In the following sample message, lines 1 and 2 are the message envelope, the next four lines are the header, and the rest of the lines are the body.

```
jdoe@siroe.com
msmith@siroe.com

Date: Tues, 1 Apr 2003 15:01 PST
From: John Doe
To: Mike Smith
Subject: Lunch today

Mike,
Just confirming our lunch appointment today I will meet you at the
restaurant at noon.
John
```

Note: As stated earlier, do not directly read from or write messages to the MTA message queues. Always use the SDK routines or Callable Send. The file structure of messages in the MTA queues are subject to change. In addition, site specific constraints may be placed on things such as encodings, and character set usage. The SDK routines automatically handle these and other issues.

Threads and Enqueue Contexts

Each individual message being enqueued to the MTA is represented within the SDK by an opaque enqueue context of type **mta_nq_t**. This enqueue context is created by mtaEnqueueStart() and destroyed by mtaEnqueueFinish(). Throughout the enqueue process, the message being enqueued is referenced through its enqueue context. A program using the SDK may simultaneously enqueue multiple messages, each message represented by its own enqueue context. Indeed, multiple threads may simultaneously enqueue one or more messages per thread. The only requirement is that a specific enqueue context not be simultaneously used by two or more threads. In the event that the SDK detects simultaneous usages, it returns the **MTA_THREAD** error.

Enqueuing Dequeued Mail

If a message being enqueued is the result of dequeuing a message, then all envelope fields can automatically be carried over from the old message to the new message. Both per-message fields (such as envelope IDs) and per-recipient fields (such as delivery receipt requests) can be preserved. This preservation is achieved by supplying the associated dequeue context to the routines mtaEnqueueStart(), or mtaEnqueueTo(), or both. Supplying the dequeue context to mtaEnqueueStart() preserves per-message envelope fields, while supplying the dequeue context to mtaEnqueueTo() preserves the per-recipient fields for the specified envelope recipient.

The following section contains information on message dequeuing and message dequeue contexts.

Dequeuing Messages

Messages stored in the MTA message queues are removed from their queues by dequeuing them. This is typically done by channel programs as mentioned in "Channel Programs and Message Queuing". When a message is dequeued, it is literally removed from the MTA message queues and, as far as the MTA is concerned, no longer exists. That is, dequeuing a message relieves the MTA of all further responsibility for the message. The responsibility is assumed to have been passed on to some other entity such as another MTA or a message store.

The channel name used by the program identifies the MTA message queue being serviced by the program. The channel name can either be explicitly specified by the program or determined from the run time environment using the **PMDF_CHANNEL** environment variable.

Note: Channel naming conventions: the name must be 32 bytes or less, should be in lower case, and if the channel will have multiple instantiations, then it should be given a generic name, such as **tcp**, and then each instantiation can be given a specific version of it, such as **tcp_local**, **tcp_auth**, **tcp_intranet**.

Multiple programs may simultaneously process the same message queue. The SDK and Job Controller will automatically coordinate such efforts, using file locks to prevent two or more programs or threads from simultaneously processing the same message. When the message processing program is called, the message to be process is locked so that no other jobs may access that message. The message is then unlocked when mtaDequeueMessageFinish() is called, or when the program exits, normally or abnormally. For more information, see "Dequeue Message Processing Routine Tasks".

Threads and Dequeue Contexts

Each individual message being dequeued from the MTA is represented within the SDK by an opaque dequeue context of type **mta_dq_t**. Each dequeue context is created by mtaDequeueStart() and passed to a caller-supplied processing procedure. Each dequeue context is then destroyed when mtaDequeueMessageFinish() is called. Throughout the dequeue process, the message being dequeued is referenced through its dequeue context. Under typical usage, a program will have multiple threads operating, each simultaneously dequeuing a message. However, it is not permitted for two threads to simultaneously use the same dequeue context in calls to the SDK. In the event the SDK detects simultaneous usages, it returns the **MTA_THREAD** error.

Message Processing Threads

When mtaDequeueStart() is called, a communication path with the MTA Job Controller is established. The Job Controller is then asked if there are messages to be processed for the channel. Typically there will be messages to process since the Job Controller normally only starts channel programs when there are queued messages in need of processing. Based upon information obtained from the Job Controller, mtaDequeueStart() will then begin to create non-joinable processing threads. Each processing thread immediately begins processing the queued messages.

For further information about the exact steps a message processing thread goes through, see "Debugging Programs and Logging Diagnostics".

String-valued Call Arguments

Strings passed as call arguments to the MTA SDK routines also have an associated length argument. Use of the length argument is optional; that is, if you do not know the length or do not wish to supply it, then supply a value of zero for the length argument. However, in that case the supplied string must be NULL terminated so that the SDK routine can determine the string's length. When a non-zero length is supplied, then the string does not need to be NULL terminated. Wherever possible, the SDK routines return pointers to output strings rather than returning the strings themselves. These pointers are always thread safe; however, when associated with an SDK context they often are only valid as long as the context itself is valid. Such limits will be noted in the description of the individual routines in "Dequeuing Messages". In some cases, an output string buffer must be supplied, as with the mtaDateTime() and mtaUniqueString() routines.

Internally, the MTA has several basic string sizes. Users of the SDK generally do not need to concern themselves with this fact. However, at times it may be helpful to be aware of them as they can provide an upper bound on the length of various strings you might encounter. As shown in Table 1–1, for instance, channel names will never be longer than **CHANLENGTH** bytes; channel option values will never exceed a length of **BIGALFA_SIZE** bytes; and envelope addresses will never exceed a length of **ALFA_SIZE** bytes.

Table 1–1 ims-ms Channel

Symbolic Names	Value in Bytes	Typical Usage
ALFA_SIZE	256	Upper limit on the length of an address
BIGALFA_SIZE	1024	Upper limit on the length of message line and channel option value
CHANLENGTH	32	Upper limit on the length of a channel name

Item Codes and Item Lists

A number of the MTA SDK routines accept a variable length list of item code arguments. For instance, mtaInit() has the call syntax:

```
int mtaInit(int item_code, ...)
```

That is to say, it accepts one or more integer-valued call arguments. These call arguments are referred to as an "item code list" or, more simply, an "item list." Each item list must be terminated by a call argument with the value **0**. As such, the call syntax for mtaInit() can be expressed as

```
int mtaInit([int item_code[, ...]], 0)
```

There can be zero or more item codes with non-zero values which must then be followed by an item code with the value zero.

In the MTA SDK, item lists serve two purposes. First, they allow code using the SDK to specify optional behaviors and actions to the SDK. Second, they provide an extension mechanism for future versions of the SDK to extend the functionality of routines through the introduction of new item codes.

However, there is a drawback to the use of item lists; the number of items passed to an SDK routine must be known at compile time. That is, it is difficult if not impossible for a program at run time to adjust the number of item codes that it wishes to pass. In recognition of this limitation, all SDK routines that accept an item code list also accept a pointer to an arbitrary length array of item codes. Such an array is referred to as an "item list array" and is specified with the **MTA_ITEM_LIST** item code. This mechanism allows programs to dynamically construct the array at run time, while still using a fixed number of arguments at compile time.

The **MTA_ITEM_LIST** item code is always followed by an additional call argument whose value is a pointer to an array of **mta_item_list_t** type elements. Table 1–2 shows that each array entry has the following five fields.

Table 1–2 mta_item_list Array Entry Fields

Fields	Description
int item_code	An item code value indicating an action to be effected. The permitted item code values are routine specific.
const void *item_address	The caller-supplied address of data to be used in conjunction with the action specified by the **item_code** field. Not all actions require use of this field.
size_t item_length	When the item code has an associated string value, this field optionally provides the length in bytes of the string, not including any NULL terminator. If a value of zero is supplied, then the string pointed at by the **item_address** field must be NULL terminated.When the item code has an associated integral value, this field supplies that value. Not all actions require the use of this field.
int item_status	Only used by **mtaSend()**. Not used by other MTA SDK routines.
const char *item_ smessage	Only used by **mtaSend()**. Not used by other MTA SDK routines.

The end of the array is signified by an array entry whose item_code field has the value zero (**MTA_END_LIST**). As an example of using **MTA_ITEM_LIST**, consider the following mtaInit() call:

```
istat = mtaInit(MTA_DEBUG_SDK, MTA_DEBUG_OS, MTA_DEBUG_MM, 4,
                MTA_DEBUG_DEQUEUE, MTA_DEBUG_DECODE, 0);
```

In the above call, the decision to enable the listed debug modes is made at compile time. Using an item list array allows the decision to be made at run time as illustrated in the following example:

```
mta_item_list_t item_list[6];
int index;

index = 0;
if (debug_sdk)
   item_list[index++].item_code = MTA_DEBUG_SDK;
if (debug_os)
   item_list[index++].item_code = MTA_DEBUG_OS;
if (debug_mm)
{
   item_list[index].item_code = MTA_DEBUG_MM;
   item_list[index++].item_length = 4;
}
if (debug_dq)
   item_list[index++].item_code = MTA_DEBUG_DEQUEUE;
```

```
if (debug_decode)
   item_list[index++].item_code = MTA_DEBUG_DECODE;
item_list[index].item_code = MTA_END_LIST;
istat = mtaInit(MTA_ITEM_CODE, item_list, 0);
```

The list of item code arguments must still be terminated with a call argument with value zero. Further, item codes may simultaneously be passed as distinct call arguments and also in item list arrays. For example:

```
mtaInit(MTA_DEBUG_SDK, MTA_ITEM_LIST, item_list1,
        MTA_INTERACTIVE, MTA_ITEM_LIST, item_list2, 0);
```

In the above, the item codes **MTA_DEBUG_SDK**, **MTA_ITEM_LIST**, **MTA_INTERACTIVE**, and **MTA_ITEM_LIST** are all explicitly passed as call arguments. Additional item codes are passed via the item list arrays **item_list1** and **item_list2**.

When processing item codes, they are processed from left to right as the call argument list is interpreted. Using the above example, mtaInit() processes **MTA_DEBUG_SDK**, then **MTA_ITEM_LIST**, **MTA_INTERACTIVE**, **MTA_ITEM_LIST**, and finally the terminating 0 call argument which terminates the item code processing. When processing the first occurrence of **MTA_ITEM_LIST**, the entries of **item_list1** are processed starting with the first entry (index 0), then the second, and so on until an entry with an item code value of **0** is encountered. The same processing occurs for **item_list2**.

If two item codes set the same underlying option or value, the last processed instance of that item code will prevail. For example, the call:

```
mtaInit(MTA_DEBUG_ENQUEUE, MTA_DEBUG_MM, 10, 0);
```

will leave the enqueue debug level set to **10**. While the **MTA_DEBUG_ENQUEUE** item code sets it to **5**, the subsequent **MTA_DEBUG_MM** item code changes the setting to **10**.

2

MTA SDK Programming Considerations

This chapter describes procedures and run time instructions for the Oracle Communications Messaging Server MTA SDK.

Running Your Enqueue and Dequeue Programs

At run time, when your program enqueues a message to, or dequeues a message from the MTA, the SDK must be able to determine the name of the MTA channel under which to perform the enqueue or dequeue. If this name cannot be determined, then the enqueue or dequeue operation will fail. Consequently, when calling mtaEnqueueStart() or mtaDequeueStart(), a channel name can be specified. Whether or not you need to specify this channel name depends upon the conditions under which your program runs. While developing your program and manually running it, you may either code the channel name into your program or specify it through your run time environment with the **PMDF_CHANNEL** environment variable. For example, to do the latter on UNIX platforms use a command of the following form:

 # PMDF_CHANNEL= channel-name **program-name**

where **channel-name** is the name of the channel and **program-name** is the name of the executable program to run.

In production, if your program will run as a master or slave channel program under the MTA Job Controller, you do not need to specify the channel name The channel name will automatically be set by the Job Controller using the **PMDF_CHANNEL** environment variable. If, however, your program will be run manually or as a server, then either the program can specify its channel name through code or using the **PMDF_CHANNEL** environment variable. For the latter, setting the environment variable is typically achieved by wrapping your executable program with a shell script. The shell script would set the environment and then invoke your program, as illustrated in the following code example:

```
#!/bin/sh

PMDF_CHANNEL=_channel-name_

PMDF_CHANNEL_OPTION=_option-file-path_

export PMDF_CHANNEL PMDF_CHANNEL_OPTION

_program-name_

exit
```

The **option-file-path** shown in the previous example is the full, absolute path to the channel's option file, if any.

A program can query the SDK to determine what channel name is being used with either the mtaChannelGetName(), mtaEnqueueInfo(), or mtaDequeueInfo() routines. The former returns the channel name the SDK will use when no other name is explicitly specified through code. The latter two return the name specifically being used with a given enqueue or dequeue context.

Note: The SDK only reads the **PMDF_CHANNEL** environment variable once per program invocation. As such, running code cannot expect to change its channel name by changing the value of the environment variable.

Debugging Programs and Logging Diagnostics

The SDK has diagnostic facilities that may help in tracking down unusual behavior. Enable SDK diagnostics in one of two ways: either when the SDK is initialized with mtaInit() or afterwards with mtaDebug(). Table 2–1 lists the diagnostics types that may be enabled through either routine.

Table 2–1 Diagnostic Types Enabled Through mtaInit() and mtaDebug()

Diagnostic Type	Description
MTA_DEBUG_SDK	Provide diagnostics whenever the SDK returns an error status
MTA_DEBUG_DEQUEUE	Provide diagnostics from the MTA low-level dequeue library
MTA_DEBUG_ENQUEUE	Provide diagnostics from the MTA low-level enqueue library
MTA_DEBUG_OS	Provide diagnostics from the MTA low-level, operating-system dependent library

All diagnostic output is written to **stdout**. In the case of a channel program, this is typically the channel's debug file. Message enqueue and dequeue activities performed through the MTA SDK (and Callable Send facility) will be logged when the channels involved are marked with the **logging** channel option.

Required Privileges

Use of the MTA SDK often requires access rights to the MTA message queues and configuration data. Indeed, were such rights not required, then any user capable of logging in to the operating system of the machine running Messaging Server could read messages out of the MTA message queues and send fraudulent mail messages. Consequently, any programs using the MTA SDK need read access to the MTA configuration, possibly including files with credentials required to bind to either the Job Controller or an LDAP server or both. Additionally, programs that will enqueue messages to the MTA need write access to the MTA message queues. Programs that will dequeue messages from the MTA need read, write, and delete access to the MTA message queues.

To facilitate this access, site-developed programs that will enqueue or dequeue messages should be owned and run by the account used for Messaging Server. The programs do not need to run as a superuser with **root** access in order to enqueue or dequeue mail to the MTA. However, it is safe to allow them to do so, if needed for concerns outside the scope of Messaging Server. For instance, if the program will be

performing other functions requiring system access rights, it needs to run as a superuser with **root** access.

Compiling and Linking Programs

This section contains information useful for compiling and linking your C programs.

Compiling

To declare the SDK routines, data structures, constant, and error codes, C programs should use the *MessagingServer_home***/include/mtasdk.h** header file.

Linking Instructions for Oracle Solaris

The linking instructions that follow are for the Oracle Solaris platform:

The table that follows shows the link command used to link a C program to the SDK:

```
% SERVER_ROOT=msg_svr_base
% cc -o program program.c \ -I$SERVER_ROOT/include \ -L$SERVER_ROOT/lib \ -lmtasdk
```

In the example, *MessagingServer_home* is the directory path to the top-level Messaging Server directory, and *program* is the name of your program.

If running the program in a standalone mode, that is, not under the Job Controller, then the **CONFIGROOT**, **INSTANCEDIR**, **IMTA_TAILOR**, and the **LD_LIBRARY_PATH** environment variables must be defined. See the **imsimta** shell script used to launch MTA programs and utilities for details.

Running Your Test Programs

This section describes the tasks that are typically required for running your test programs that enqueue or dequeue messages. The tasks are divided into two groups, those used to run your test programs in a fully functional messaging environment, and those needed if you want to run them manually:

- To Run Test Programs in a Messaging Environment
- To Manually Run Your Test Programs

To Run Test Programs in a Messaging Environment

1. Add a test channel to the bottom of the **imta.cnf** file. For example:

   ```
   (required blank line)
   x_test
   x-test-daemon
   ```

2. Add rewrite rules to the top of the **imta.cnf** file. The following code fragment illustrates this:

   ```
   x_test $U%x-test@x-test-daemon
   ```

3. To enable your test channel so that mail can be addressed to **user@x_test**, recompile your configuration and restart the SMTP server. Use the instructions found in the following code example:

   ```
   imsimta cnbuild
   imsimta restart dispatcher
   ```

4. Create the **job_controller.site** text file. The file should be owned by the Messaging Server and reside in the same directory as the **job_controller.cnf** file. The following code example shows the lines you must add to the file:

   ```
   [CHANNEL=x_test]
   master_command=_file-path_
   ```

   In the above example, **file-path** is the full path to your executable program.

5. Make sure your executable has permissions and ownership such that the Messaging Server can run it.

6. Restart the Job Controller. Use the command found in the following code example:

   ```
   imsimta restart job_controller
   ```

   If the program performing enqueues is also a channel that will be dequeuing messages, and more specifically, is doing intermediate processing that leaves the envelope recipient addresses unchanged, then special rewrite rules must be used to prevent a message loop in that the channel just enqueues the mail back to itself. For directions on how to prevent a message loop and other specific examples of rewrite rules, see "Preventing Mail Loops when Re-enqueuing Mail".

To Manually Run Your Test Programs

1. If the program does not explicitly set the channel name, then you must define the **PMDF_CHANNEL** environment variable. The value of that variable must be the name of your channel. The following example shows how to set the **PMDF_CHANNEL** environment variable:

   ```
   # PMDF_CHANNEL=x_test
   # export PMDF_CHANNEL
   ```

   For further information, see "Running Your Enqueue and Dequeue Programs".

2. Ensure that any environment variables required to run a program linked against the MTA SDK are defined. See "Compiling and Linking Programs" for additional information.

3. Under some circumstances, it might be useful to comment out the **master_command=** line in the **job_controller.site** file. If you do this, you can enqueue mail to your test channel but not have the Job Controller actually run your channel program.

4. When repeatedly testing your channel program, it is often necessary to restart the Job Controller before each manual test run. Otherwise, the Job Controller will hand off messages to your program on the first manual run but not the second manual run. The Job Controller will think that retries of the messages need to be delayed by several hours. By restarting the Job Controller, you cause it to "forget" which queued messages are to be deferred. Thus, when you run your channel again, it will be presented with all of the queued messages.

Preventing Mail Loops when Re-enqueuing Mail

This section shows how to add a new rewrite rule to prevent a message loop from happening if the program is doing intermediate processing that leaves the envelope recipient addresses unchanged. Otherwise, the channel would enqueue the mail back to itself.

For discussion purposes, suppose that the channel is to provide intermediate processing for mail addressed to **user@siroe.com**. Further, the **imta.cnf** file has the following rewrite rule for **siroe.com**:

```
siroe.com $U@siroe.com
```

For example, as shown in the code example that follows, assume that the intermediate processing channel's name is "xloop_test." Near the bottom of the **imta.cnf** file with other channel definitions, you would see the following definition:

```
xloop_test
x-looptest-daemon
```

Next, a new rewrite rule for **siroe.com** needs to be added to the top of the **imta.cnf** file:

```
siroe.com $U%siroe.com@x-looptest-daemon$Nxloop_test
siroe.com $U@siroe.com
```

The new rewrite rule causes the following:

- When a new inbound or outbound message for **user@siroe.com** is enqueued to the **xloop_text channel**, it processes the message and re-enqueues it to **user@siroe.com**.
- In the new rewrite rule, **$N** says that the first (new) rewrite rule is to be ignored when the **xloop_test** channel itself enqueues a message.
- Therefore, after the **xloop_test** channel does its processing and re-enqueues the message to **user@siroe.com**, the first (new) rewrite rule is ignored and the second (old) rule is then applied, causing the message to be routed as it would have been before the **xloop_test** channel was added to the MTA.

Miscellaneous Programming Considerations

This section covers miscellaneous topics of interest to programmers using the SDK:

- Retrieving Error Codes
- Writing Output From a Channel Program
- Considerations for Persistent Programs

Retrieving Error Codes

With few exceptions, all routines in the SDK return an integer-valued result with a value of zero (0) indicating success. When a non-zero value is returned, it is also saved in a per-thread data section, which may be retrieved with either the **mtaErrno()** function or the **mta_errno** C pre-processor macro.

The exceptional routines either return nothing (that is, always succeed), or return a string pointer, and signify an error with a return value of **NULL**.

Writing Output From a Channel Program

The C runtime library **stdout** input-output destination may be usurped by the SDK, depending upon the context under which a channel program has been invoked. As such, programs that will operate as channels should use the **mtaLog()** routine to write information to their log file. Such programs should never write output directly to **stdout** or **stderr** or other generic I/O destinations, such as Pascal's **output**, or FORTRAN's default output logical unit. There is no telling where such output might

go: it might go to the Job Controller's log file, it might even go down a network pipe to a remote client or server.

> **Note:** The channel log file is a different file from the MTA log file. The mtaLog() and mtaAccountingLogClose() are unrelated routines.

Considerations for Persistent Programs

There are two main problems to consider when creating programs that persist over long periods of time (weeks or months):

- Refreshing Stale Configuration Information
- Keeping the Log File Available For Update

Refreshing Stale Configuration Information

Some programs, once started, run indefinitely. An example of this kind of program is a server that listens continually for incoming mail connections, enqueuing received messages. Site-specific configuration information is loaded at initialization. In the case of these long running programs, the information can become stale due to changes to configuration information, such as rewrite rules or channel definitions. Subsequent calls to mtaInit() do not accomplish this task. A program must exit and restart in order to ensure that all configuration information is reloaded.

Keeping the Log File Available For Update

A program that enqueues and dequeues messages may open the MTA log file, **mail.log_current**. For persistent programs, care should be taken that this log file is not left open during periods of inactivity. Otherwise, activities that require exclusive access to this file will be blocked. Before going idle, persistent programs should call mtaAccountingLogClose(). The log file will automatically reopened when needed.

> **Note:** The MTA log file, **mail.log_current**, is not the log written to by mtaLog().

3

Enqueuing Messages

This chapter describes how to use the Oracle Communications Messaging Server MTA SDK to construct a mail message and then submit the message to the MTA, referred to as "enqueuing a message."

About Enqueing Messages

The Oracle Communications Messaging Server MTA SDK provides routines with which to construct a mail message and then submit the message to the MTA. The MTA then effects delivery of the message to its recipients. The act of submitting a message to the MTA for delivery is referred to as "enqueuing a message." This choice of terminology reflects the fact that each message submitted to the MTA for delivery is placed into one or more message queues. Using its configuration, the MTA determines how to route each message to its destination and which message queues to place each the message into. However, programs enqueuing messages do not need to concern themselves with these details; they merely supply the message's list of recipients and the message itself. The recipients are specified one-by-one as RFC 2822 conformant Internet email addresses. The message header and content is supplied in the form of an RFC 2822 and MIME conformant email message.

When starting a coding project to enqueue messages to the MTA, always stop to consider whether simply using SMTP will be acceptable. The advantage of using SMTP is that it will work with any MTA SMTP server, making it portable. The disadvantages are poorer performance and lack of flexibility and control.

Basic Steps to Enqueue Messages

The basic steps necessary to enqueue one or more messages to the MTA are:

1. Initialize SDK resources and data structures with mtaInit().
2. For each message to enqueue:
    1. Specify the message envelope with mtaEnqueueStart() and mtaEnqueueTo().
    2. Specify the message header with mtaEnqueueWrite() or mtaEnqueueWriteLine().
    3. Optionally, if a message body is to be supplied, terminate the message header and start the message body by writing a blank line to the message with mtaEnqueueWrite() or mtaEnqueueWriteLine().
    4. Optionally if a message body is to be supplied, write the message body with mtaEnqueueWrite() or mtaEnqueueWriteLine().
    5. Submit the message with mtaEnqueueFinish().

3. When you have completed enqueuing messages, deallocate SDK resources and data structures with mtaDone().

In Step 2e, mtaEnqueueFinish() commits the message to disk. As part of the enqueue process, the MTA performs any access checks, size checks, format conversions, address rewritings, and other tasks called for by the site's MTA configuration. After these steps are completed and the message has been successfully written to disk, mtaEnqueueFinish() returns.

Other MTA processes controlled by the MTA Job Controller then begin processing the new message so as to effect its delivery. In fact, these processes may begin handling the new message before mtaEnqueueFinish() even returns. As such, mtaEnqueueFinish() doesn't block waiting on these processes; it returns as soon as all requisite copies of the enqueued message have been safely written to disk. The subsequent handling of the newly enqueued message is performed by other MTA processes, and the program which enqueued the message isn't left waiting for them.

A message submission can be aborted at any point in Step 2 by calling either mtaEnqueueFinish() with the **MTA_ABORT** option specified or **mtaDone()**. Using the first method, mtaEnqueueFinish() aborts only the specified message enqueue context while allowing additional messages to be enqueued. Whereas, mtaDone() aborts all active message enqueue contexts in all threads, and deallocates SDK resources disallowing any further submission attempts until the SDK is again initialized.

Originating Messages

Messages enqueued to the MTA fall into one of two broad classes: new messages being originated and messages which were originated elsewhere and which are being transferred into the MTA. The former are typically the product of a local user agent or utility which uses the MTA SDK. The latter are generated by remote user agents, and received by local programs such as SMTP or HTTP servers which then enqueue them to the MTA for routing or delivery or both. In either case, it is the job of the MTA to route the message to its destination, be it a local message store or a remote MTA.

The only distinction the MTA SDK makes between these two cases occurs when the message's recipient addresses are specified. For new messages being originated, the recipient addresses should be added to both the message header and its envelope. For messages originated elsewhere, the recipient addresses should only be added to the message's envelope. For a discussion of messages originated elsewhere, see "Transferring Messages into the MTA", and "Intermediate Processing Channels".

When originating a new message, it is easiest to use the **MTA_TO**, **MTA_CC**, and **MTA_BCC** item codes with mtaEnqueueTo(). That tells the SDK to use the specified addresses as both the envelope recipient list and to put them into the message's header. When using this approach, do not specify any **From:**, **To:**, **Cc:**, or **Bcc:** header lines in the supplied message header; the SDK will add them automatically.

An example of using this approach is found in the following section.

A Simple Example of Enqueuing a Message

The program shown in "Example 3-1 Enqueuing a Message" demonstrates how to enqueue a simple "Hello World" message. The originator address associated with the message is that of the MTA postmaster. The recipient address can be specified on the invocation command line.

After the Messaging Server product is installed, this program can be found in the following location:

MessagingServer_home/**examples/mtasdk/**

Note that certain lines of code have numbered comments immediately preceding them of the format:

```
/* This generates output line N */
```

where *N* corresponds to the numbers found next to certain output lines in the sample output in "Enqueuing a Message Example Output".

Refer to "Running Your Test Programs" for information on how to run the sample program.

Example 3-1 Enqueuing a Message

```
/* hello_world.c -- A simple "Hello World!" enqueue example */
#include &lt;stdio.h\>
#include &lt;stdlib.h\>
#include "mtasdk.h"

mta_nq_t *ctx = NULL;
static void quit(void);
#define CHECK(x) if(x) quit();

void main(int argc, const char *argv[])
{
     char buf[100];

     /* Initialize the SDK */
     CHECK(mtaInit(0));

     /* Start a new message; From: postmaster*/
     /* This generates output line 1 */
     CHECK(mtaEnqueueStart(&amp;ctx, mtaPostmasterAddress(NULL, NULL,
                          0), 0, 0));

     /* Enqueue the message to argv[1] or root */
     /* This generates output line 2 */
     CHECK(mtaEnqueueTo(ctx, (argv[1] ? argv[1] : "root"), 0, 0));

     /* Date: header line */
     /* This generates output line 3 */
     CHECK(mtaEnqueueWriteLine(ctx, "Date: ", 0, mtaDateTime(buf,
                               NULL, sizeof(buf), 0), 0, NULL))

     /* Subject: header line */
     /* This generates output line 4 */

     CHECK(mtaEnqueueWriteLine(ctx, "Subject: " __FILE__, 0,
                               NULL));

     /* Blank line ending the header, starting the message body */
     /* This generates output line 5 */
     CHECK(mtaEnqueueWriteLine(ctx, "", 0, NULL));

     /* Text of the message body (2 lines) */
     /* This generates output line 6 */
     CHECK(mtaEnqueueWriteLine(ctx, "Hello", 0, NULL));
     /* This generates output line 7 */
     CHECK(mtaEnqueueWriteLine(ctx, " World!", 0, NULL));

     /* Enqueue the message */
```

```
    CHECK(mtaEnqueueFinish(ctx, 0));

    /* All done */
    mtaDone();
}

void quit(void)
{
    fprintf(stderr, "The MTA SDK returned the error code %d\n
            %s", mta_errno, mtaStrError(mta_errno, 0));
    if (ctx)
        mtaEnqueueFinish(ctx, MTA_ABORT, 0);
    exit(1);
}
```

Enqueuing a Message Example Output

Table 3–1 shows the output generated by the enqueuing example. Comment numbers correspond to the numbered comments in "Example 3-1 Enqueuing a Message".

Table 3–1 Enqueuing Example Output Comments

Comment Number	Output Lines
none	**Received:from siroe.com by siroe.com (SunONE Messaging Server 6.0)id<01GP37SOPRW0A9KZFV@siroe.com\>; Fri, 21 Mar 2003 09:07:32 -0800(PST)**
3	**Date: Fri, 21 Mar 2003 09:07:41 -0800 (PST)**
1	**From: postmaster@siroe.com**
2	**To: root@siroe.com**
4	**Subject: enqueuing_example.c Message-id: <01GP37SOPRW2A9KZFV@siroe.com\> Content-type: TEXT/PLAIN; CHARSET=US-ASCII Content-transfer-encoding: 7BIT**
5	missing value
6	**Hello**
7	**World!**

Transferring Messages into the MTA

When transferring a message originated elsewhere into the MTA, programs should use the **MTA_ENV_TO** item code with mtaEnqueueTo(). This way, each of the recipient addresses will only be added to the message's envelope, and not to its already constructed header. Additionally, supply the message's header as-is. Do not remove or add any origination or destination header lines unless necessary. Failure to use the **MTA_ENV_TO** item code will typically cause the SDK to add **Resent-** header lines to the message's header.

"A Complex Dequeuing Example", and "A Simple Virus Scanner Example" both illustrate the use of the **MTA_ENV_TO** item code.

Intermediate Processing Channels

Like programs which transfer messages into the MTA, intermediate processing channels should also use the **MTA_ENV_TO** item code with mtaEnqueueTo(). When

re-enqueuing a message, intermediate processing channels should also preserve any MTA envelope fields present in the message being re-enqueued. This is done using the **MTA_DQ_CONTEXT** item code in conjunction with mtaEnqueueStart() and mtaEnqueueTo(). Failure to preserve these envelope fields can result in loss of delivery receipt requests, special delivery flags, and other flags which influence handling and delivery of the message.

"A Complex Dequeuing Example" and "A Simple Virus Scanner Example" both illustrate the use of the **MTA_ENV_TO** and **MTA_DQ_CONTEXT** item codes. item codes. Both of those examples represent intermediate processing channels that handle previously constructed messages. As such, they do not need to alter the existing message header, and they preserve any MTA envelope fields.

Delivery Processing Options (Envelope Fields)

A variety of delivery processing options may be set through the MTA SDK. These options are then stored in the message envelope and are generically referred to as "envelope fields". Options which pertain to the message as a whole are set with mtaEnqueueStart(). Options which pertain to a specific recipient of the message are set with mtaEnqueueTo(). Table 3–2 shows these options, per message and per recipient.

Table 3–2 Delivery Processing Options

Option	Description
Delivery flags	These flags are used to communicate information between channels. For instance, a scanning channel might set the flag to indicate suspected spam content. A delivery channel could then see that the flag is set and, at delivery time, add a header line indicating potential spam content. These flags may also be set using the **deliveryflags** channel option.
Notification flags	These flags influence whether delivery or non-delivery notification messages are generated. They can be set on a per recipient basis. Typically, they are used to request a delivery receipt. Another common usage is for bulk mail to request no notifications, neither delivery nor non-delivery.
Original recipient address	This field is specified on a per recipient basis. It is used to indicate the original form of the associated recipient's address. This original address can then be used in any notification messages. Its use allows the recipient of the notification to see the original address they specified rather than its evolved form. For example, the recipient would see the name of the mailing list they posted to rather than the failed address of some member of the list.
Envelope ID	Set on a per message basis, this is an RFC 1891 envelope ID and can appear in RFC 1892 - 1894 conformant notifications about the message.
Fragmentation size	Set on a per message basis, this controls if and when the message is fragmented into smaller messages using the MIME **message/partial** mechanism.

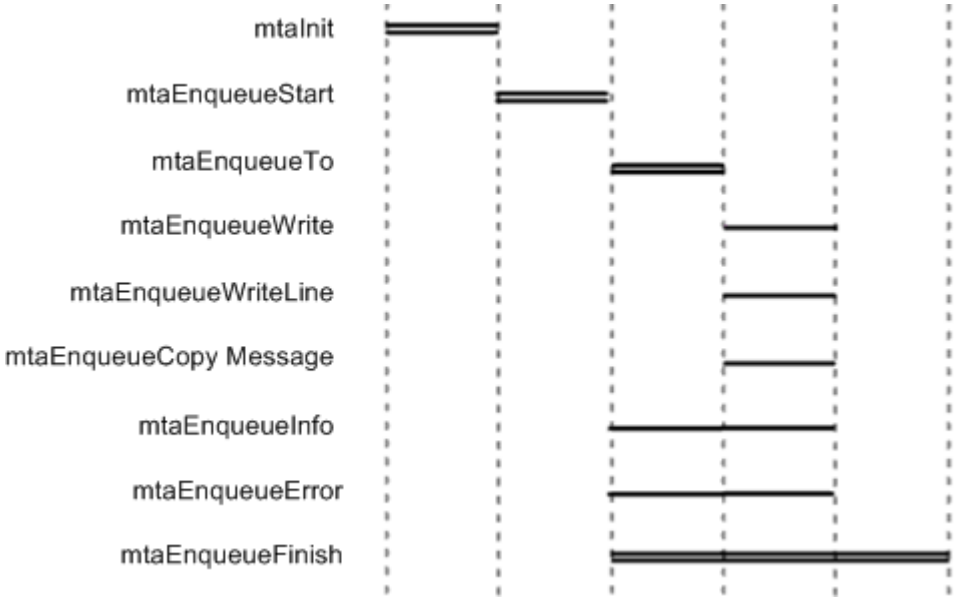
For additional information, see the descriptions of mtaEnqueueStart(), and mtaEnqueueTo().

Order Dependencies

When you are constructing programs, there is a calling order for the MTA SDK routines that must be observed. For a given enqueue context, some routines must be called before others.

Figure 3–1 depicts the calling order dependency of the message enqueue routines. To the right of each routine name appears a horizontal line segment, possibly broken across a column, for example, **mtaEnqueueWrite()**. Routines for which two horizontal line segments, one atop the other, appear are required routines; that is, routines that must be called in order to successfully enqueue a message. These routines are mtaDequeueStart(), mtaEnqueueTo(), and mtaEnqueueFinish(). To determine at which point a routine may be called, start in the leftmost column and work towards the rightmost column. Any routine whose line segment lies in the first (leftmost) column may be called first. Any routine whose line segment falls in the second column may next be called, after which any routine whose line segment falls in the third column may be called, and so forth. When more than one routine appears in the same column, any or all of those routines may be called in any order. Progression from left to right across the columns is mandated by the need to call the required routines. Of the required routines, only mtaEnqueueTo() may be called multiple times for a given message.

Figure 3–1 Calling Order Dependency for Message Enqueue Routines



4

Dequeuing Messages

This chapter describes how to use the Oracle Communications Messaging Server MTA SDK to dequeue messages.

About Dequeing Messages

Once enqueued to the MTA, messages are processed by using the SDK dequeue routines. These routines provide channel programs and MTA utilities with programmatic access to queued messages. With these routines, a channel program can process its queue of messages, accessing the message's envelope information and message content.

How Dequeuing Works

Channel programs wishing to dequeue messages from the MTA must associate themselves with a specific MTA channel or channels. Without this information, the MTA SDK does not know which channel queue to draw messages from. This information can be provided implicitly with the **PMDF_CHANNEL** environment variable, or explicitly by specifying the name of the MTA channel to process when calling mtaDequeueStart().

The dequeue process is initiated by calling the routine mtaDequeueStart(). A key piece of required information passed to mtaDequeueStart() is the address of a caller-supplied routine designed to process a single message. This routine will be repeatedly called by mtaDequeueStart() until there are no more queued messages in need of processing. One call is made per message to be processed.

Unless otherwise instructed, mtaDequeueStart() will use multiple threads of execution to process queued messages. Each thread of execution will repeatedly invoke the caller-supplied routine, once for each message to be processed. Thus, by default the caller-supplied routine is expected to be "thread-safe." That is, it is expected to support being called simultaneously by more than one thread of execution. If the caller-supplied routine is not thread safe, then mtaDequeueStart() can be instructed to use a single thread of execution, as illustrated in "A Complex Dequeuing Example".

Basic Dequeuing Steps

The following basic steps are necessary to dequeue messages:

1. Initialize SDK resources and data structures with mtaInit().
2. Call mtaDequeueStart(), passing it the address of the caller-supplied routine that is to be used to process each message. When mtaDequeueStart() is called, it does not

return until all queued messages requiring processing have been processed, thus blocking the thread calling it until it is finished.

3. For each queued message requiring processing, an execution thread created by mtaDequeueStart() calls the routine whose address was provided in Step 2. Threads created by mtaDequeueStart()mtaDequeueStart() each sequentially process multiple messages. That is, **mtaDequeueStart()** does not create a distinct thread for each and every queued message to be processed. For a list of the tasks the processing routine should do, see "Caller-Supplied Processing Routine".

   Note: The **mtaDequeueStart()** routine will use one or more threads, with each thread calling the message processing routine. The maximum number of threads allowed can be set when calling mtaDequeueStart(). Consequently, a program that does not support threading should specify a maximum of one thread when it calls mtaDequeueStart().

   For a list of the tasks the processing routine should do, see "Dequeue Message Processing Routine Tasks".

4. After mtaDequeueStart() returns, deallocate SDK resources and data structures with a call to mtaDone().

Caller-Supplied Processing Routine

Channel programs typically perform some form of processing on each message they dequeue. For instance, virus scanning, MMS conversion, decryption, delivery to a proprietary messaging system, and so forth. When using the MTA SDK, channel programs must provide a routine which initiates this processing on a per message basis. That is, programs must supply a routine that to be called to process a single queued message. Throughout the rest of this text, this caller-supplied routine will be referred to as "the caller-supplied processing routine" or, for short, "the processing routine."

When called by one of the mtaDequeueStart() execution threads, the processing routine uses the SDK to access the message's envelope, header, and any content. Upon completion of processing, the message is then either removed from the MTA queues, or, in the event of a temporary error, left in its queue for a later processing attempt.

Dequeue Message Processing Routine Tasks

The processing routine processes a single queued message per invocation. The specific steps that a processing routine takes are:

1. Read the envelope recipient list with repeated calls to mtaDequeueRecipientNext(). When mtaDequeueRecipientNext() returns the **MTA_EOF** status code, the list has been exhausted and all envelope recipient addresses have been provided. All queued messages are guaranteed by the MTA to always have at least one envelope recipient address.

2. Read the message, both header and body, with repeated calls to mtaDequeueLineNext(). When mtaDequeueLineNext() returns the **MTA_EOF** status code, the message has been exhausted; that is, there is no more message text to retrieve. The message will be an RFC 2822 conformant message. As such, the division between the message's header and content will be demarked by a blank

line (a line with a length of zero). A message may have no content; that is, a message may have just a header.

3. Process the message. The processing in this step could be almost anything, including possibly enqueuing a new message or messages with the MTA SDK. The details of this step will depend upon the purpose of the program itself. Programs needing to do MIME parsing should consider using the mtaDecodeMessage() routine. For further information about message processing threads and caller-supplied message processing routines, see "Processing the Message Queue".

4. Report the disposition of each envelope recipient with per recipient calls to mtaDequeueRecipientDisposition(), or a single call to **mtaDequeueMessageFinish()** with the **MTA_DISP** item code. Table 4–1 lists the valid recipient dispositions.

Table 4–1 mtaDequeueRecipientDisposition() Valid Recipient Dispositions

Symbolic Name	Description
MTA_DISP_DEFERRED	Unable to process this recipient address. Processing has failed owing to a temporary problem, such as the network is down, a remote host is unreachable, or a mailbox is busy. Retry delivery for this recipient at a later time as determined by the configuration of the channel.
MTA_DISP_DELIVERED	Recipient address successfully delivered. Generate a delivery status notification if required.
MTA_DISP_FAILED	Unable to process this recipient address. Processing has failed owing to a permanent problem, such as an invalid recipient address, or recipient over quota. No further delivery attempts should be made for this recipient. Generate a non-delivery notification if required.
MTA_DISP_RELAYED	Recipient address forwarded to another address or sent into a non-RFC 1891 (NOTARY) mail system. The message's NOTARY information was, however, preserved. There is no need to generate a relayed notification message.
MTA_DISP_RELAYED_ FOREIGN	Recipient address forwarded to another address or gatewayed to a non-RFC 1891 (NOTARY) mail system; the messages NOTARY information was not preserved; generate a relayed notification message if required.
MTA_DISP_RETURN	For this recipient, return the message as undeliverable. Generate a non-delivery notification if required. This disposition is intended for use by queue management utilities. It is not intended for channel programs.
MTA_DISP_TIMEDOUT	Unable to process this recipient address. Processing failed due to timing out. This disposition is intended for use by the MTA Return Job. Channel programs should not use this disposition.

5. Dequeue the message with **mtaDequeueMessageFinish()**. The message is not actually removed from the channel queue until this final step. This helps ensure that mail is not lost should the channel program fail unexpectedly, or some other unexpected disaster occurs. When this routine is called, the resulting processing depends on the disposition of the envelope recipient addresses reported with mtaDequeueRecipientDisposition() (see Step 4 in this task list). If all recipients have a permanent disposition (all of the ones listed in the previous table, except **MTA_DISP_DEFERRED**), then any required non-delivery notifications are generated and the message is permanently removed from the MTA queue. If all recipients are to be deferred (**MTA_DISP_DEFERRED**), then no notifications are generated and the message is left in the queue for later delivery attempts. If,

however, some recipients have a permanent disposition and others are deferred, then the following happens:

a. Notifications are generated for those recipients with permanent dispositions that require notifications.

b. A new message is enqueued for just the deferred recipients.

c. The original message is removed from the queue. Deferred messages will not be processed by this routine more than once, unless another delivery attempt is made for the deferred message while the process is still running. How long a message is deferred is configured as part of a channel's definition, using the **backoff** channel option.

6. When finished, the processing routine should return with a status code of zero (**0**) to indicate a success, and an appropriate **MTA_** status code in the event of an error. If the processing routine returns before calling **mtaDequeueFinish()**, then the message that was being handled is left in its queue for a subsequent processing attempt. It will be as if the **MTA_DISP_DEFFERED** disposition was set for all of the message's recipients. This will be the case even if the processing routine returns a success status code of zero. In the event that the processing routine needs to abort processing of a single message, it should call **mtaDequeueMessageFinish()** with the **MTA_ABORT** flag set. If the processing routine returns with a status code of **MTA_ABORT**, then the execution thread that called the processing routine will perform an orderly exit. Consequently, the program can prematurely terminate itself in a graceful fashion by causing its processing routine to begin returning the **MTA_ABORT** status code each time it is called.

The process_message() Routine

This caller-supplied routine is invoked by the processing threads to do the actual processing of the messages.

The following code example shows the required syntax for a **process_message()** routine:

```
int process_message(void **ctx2, void *ctx1, mta_dq_t *dq_ctx,
                    const char *env_from, int env_from_len);
```

Table 4–2 lists the required arguments for a **process_message** routine, and gives a description of each.

Table 4–2 Process_Message Routine Arguments

Arguments	Description
ctx2	A writable pointer that the **process_message()** routine can use to store a pointer to a per-thread context. See the description that follows for further details.
ctx1	The caller-supplied private context passed as **ctx1** to mtaDequeueStart().
dq_ctx	A dequeue context created by mtaDequeueStart() and representing the message to be processed by this invocation of the **process_message()** routine.
env_from	A pointer to the envelope **From:** address for the message to be processed. Since Internet messages are allowed to have zero length envelope **From:** addresses, this address can have zero length. The address will be NULL terminated.

Table 4–2 (Cont.) Process_Message Routine Arguments

Arguments	Description
env_from_len	The length in bytes of the envelope **From:** string. This length does not include any NULL terminator.

When a processing thread first begins running, it sets the value referenced by **ctx2** to NULL. This assignment is made only once per thread and is done before the first call to the **process_message()** routine. Consequently, on the first call to the **process_message** routine by a given execution thread, the following test is true:

```
*ctx2 == NULL
```

That test will remain true until such time that the **process_message()** routine itself changes the value by making an assignment to ***ctx2**. If the **process_message()** routine needs to maintain state across all calls to itself by the same processing thread, it can allocate memory for a structure to store that state in, and then save a pointer to that memory with **ctx2**. The following code snippet demonstrates this:

```
int process_message(void **ctx2, void *ctx1, const char *env_from,
                    size_t env_from_len)
{
    struct our_state_t *state;

    state = (our_state_t *)(*ctx2);
    if (!state)
    {
        /*
         * First call for this thread.
         * Allocate a structure in which to store the state
         * information
         */
        state = (our_state_t *)calloc(1, sizeof(our_state_t));
        if (!state) return(MTA_ABORT);
        *ctx2 = (void *)state;

        /*
         * Set any appropriate initial values for the state
         * structure
         */
       ...
    }
...
```

For a sample **process_message()** routine, see the example code in the section that follows.

A Simple Dequeuing Example

The program shown in "Example 4-1 A Simple Dequeue" constitutes a simplified batch-SMTP channel that reads messages from a message queue, converting each message to batch SMTP format, and writes the result to **stdout**. If the conversion is successful, then the message is dequeued, otherwise it is deferred.

Some lines of code are immediately preceded by a comment of the format:

```
/* See exmplanatory comment N */
```

where *N* is a number.

The numbers are links to some corresponding explanatory text in the section that follows this code, see "Explanatory Text for Numbered Comments in the Simple Dequeue Example". Find the sample output in "Output from the Simple Dequeue Example".

Example 4-1 A Simple Dequeue

```
/*  dequeue_simple.c -- A simple dequeue example: write BSMTP to stdout
 */
#include &lt;stdio.h>
#include &lt;stdlib.h>
#include "mtasdk.h"

static mta_dq_process_message_t process_message;

int main()
{
     int ires;

     /*
      * Initialize the MTA SDK
      */
     if ((ires = mtaInit(0)))
     {
         mtaLog(mtaInit() returned %d; %s\n, ires,
                mtaStrError(ires, 0));
         return(1);
     }

     /*
      *  Start the dequeue loop.  Since this example uses stdout
      *  for output, we indicate that we only support a single
      *  thread:
      *  (MTA_THREAD_MAX_THREADS = 1).
      */
     /*  See explanatory comment 1  */
     ires = mtaDequeueStart(NULL, process_message, NULL,
                            MTA_THREAD_MAX_THREADS, 1, 0);

     /*
      *  Check the return status
      */
     /*  See explanatory comment 2  */
     if (!ires)
          /* Success */
          return(0);

     /*
      *  Print an error message to stderr
      */
     /*  See explanatory comment 3  */
     mtaLog("mtaDequeueStart() returned %d; %s\n", ires,
             ires, mtaStrError(ires, 0));

     /* And exit with an error */
     return(1);
}

/*  See explanatory comment 4  */
static int process_message(void **my_ctx_2, void *my_ctx_1,
                           mta_dq_t *dq, const char *env_from,
```

```
                              size_t env_from_len)
{
     int ires;
     const char *to, *line;
     size_t len;

     /*  See explanatory comment 5  */
     if (!(*my_ctx_2))
     {
          *my_ctx_2 = (void *)1;
          printf("HELO\n");
     }
     else
          printf("RSET\n");

     /*  Output the command:
      *     MAIL FROM: &lt;from-adr>
      */
     printf("MAIL FROM:&lt;%s>\n", env_from);

     /*
      *  Output the command:
      *     RCPT TO: &lt;to-adr>
      *  for each recipient address
      */
     /*  See explanatory comment 6  */
     while (!(ires = mtaDequeueRecipientNext(dq, &amp;to,
                                             &amp;len, 0)))
     {
          printf("RCPT TO:&lt;%s>\n", to);
          /*  See explanatory comment 7  */
          mtaDequeueRecipientDisposition(dq, to, len,
                                         MTA_DISP_DELIVERED, 0);
     }

     /*
      *  If ires == MTA_EOF, then we exited the loop normally;
      *  otherwise, theres been an error of some sort.
      */
     if (ires != MTA_EOF)
          /*  See explanatory comment 8  */
          return(ires);
     /*
      *  Now output the message itself
      */
     printf("DATA\n");
     /*  See explanatory comment 9  */
     while (!(ires = mtaDequeueLineNext(dq, &amp;line, &amp;len)))
         /*  See explanatory comment 10  */
         printf("%.*s\n", len, line);

     /*
      *  If ires == MTA_EOF, then we exited normally;
      *  otherwise, theres been an error of some sort.
      */
     if (ires != MTA_EOF)
         /*  See explanatory comment 8  */
         return(ires);

     /*
```

```
     *  Output the . command to terminate this message
     */
    printf(".\n");

    /*
     *  And dequeue the message
     */
    /*  See explanatory comment 11  */
    ires = mtaDequeueMessageFinish(dq, 0);

    /*
     *  All done; return ires as our result
     */
    /*  See explanatory comment 12  */
    return(ires);
}
```

Explanatory Text for Numbered Comments in the Simple Dequeue Example

The numbered explanatory text that follows corresponds to the numbered comments in "Example 4-1 A Simple Dequeue":

1. To start the dequeue processing, mtaDequeueStart() is called, and it calls **process_message()**, which processes each queued message. Since **process_message()** uses **stdout** for its output, only one message can be processed at a time. To effect that behavior, mtaDequeueStart() is called with the **MTA_THREAD_MAX_THREADS** set to one.
2. If the call to mtaDequeueStart() succeeds, the program exits normally.
3. If the call to mtaDequeueStart() fails, a diagnostic error message is displayed and the program exits with an error status.
4. **process_message()** is called by mtaDequeueStart() for each queued message.
5. The private context in **process_message()** tracks whether or not this is the first time the routine has been called. On the first call, the memory pointed at by **my_ctx_2** is guaranteed to be **NULL**.
6. The routine obtains each envelope recipient address, one at a time, using calls to **mtaDequeueRecipientNext()**.
7. Each recipient is marked as delivered using mtaDequeueRecipientDisposition(). An actual channel program would typically not make this call until after processing the message further.
8. If **process_message()** returns without first dequeuing the message, mtaDequeueStart() defers the message for a later delivery attempt.
9. The routine calls **mtaDequeueLineNext()** to read the message header and body, one line at a time. When there are no more lines to read, **mtaDequeueLineNext()** returns a status of **MTA_EOF**. When a line is read successfully, **mtaDequeueLineNext()** returns a status of **MTA_OK**.
10. The lines returned by **mtaDequeueLineNext()** might not be **NULL** terminated because the returned line pointer might reference a line in a read-only, memory-mapped file.
11. Once the message has been processed and all the disposition of all recipients set, **mtaDequeueMessageFinish()** is called. This actually dequeues the message.
12. When all message processing is complete, **process_message()** exits. It is called again for each additional message to be processed.

Output from the Simple Dequeue Example

```
HELO
MAIL FROM:&lt;sue@siroe.com\>
RCPT TO:&lt;dan@siroe.com\>
DATA
Received:from siroe.com by siroe.com (SunONE Messaging Server 6.0)id
 &lt;01GP37SOPRW0A9KZFV@siroe.com\>; Fri, 21 Mar 2003 09:07:32 -0800(PST)
Date: Fri, 21 Mar 2003 09:07:41 -0800 (PST)
From: postmaster@siroe.com
To: root@siroe.com
Subject: mtasdk_example1.c
Message-id: &lt;01GP37SOPRW2A9KZFV@siroe.com\>
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT

Hello
  world!
.
QUIT
```

Processing the Message Queue

This section describes the steps undertaken by each execution thread created by mtaDequeueStart(). Each execution thread processes a subset of the channel's queued messages by repeatedly calling the caller-supplied processing routine, **process_message()**.

To process queued messages, a processing thread takes the following steps:

1. The thread sets **ctx2** to have the value NULL:**ctx2 = NULL;** For information on the **process_message()** arguments, see "The process_message() Routine".
2. The execution thread communicates with the Job Controller to obtain a message file to process. If there are no more message files to process, then go to Step 9.
3. For the message file, the execution thread creates a dequeue context that maintains the dequeue processing state for that message file.
4. The execution thread then invokes the caller-supplied **process_message()** routine, passing to it the dequeue context created in "Processing the Message Queue", as shown in the example that follows **istat = process_message(&ctx2, ctx1, &dq_ctx, env_from, env_from_len);** For information on the call arguments for **process_message()**, see "The process_message() Routine".
5. The **process_message()** routine then attempts to process the message, ultimately removing it from the channel's queue, or leaving the message file for a later processing attempt.
6. If **mtaDequeueMessageFinish()** was not called before **process_message()** returned, then the queued message is deferred. That is, its underlying message file is left in the channel's queue and a later processing attempt is scheduled.
7. The dequeue context is destroyed.
8. If the **process_message()** routine did not return the **MTA_ABORT** status code, then repeat this cycle starting at Step 2.
9. If a caller-supplied **process_done()** routine was passed to **mtaDequeueStart()**, it is called now, for example:

10. **process_done(&ctx2, ctx1);** Through "The process_done() Routine", the program can perform any cleanup necessary for the execution thread. For example, freeing up any private context and associated resources stored in the **ctx2** call argument. For a description of the **process_done()** routine, see "The process_done() Routine".
11. The thread exits. For an example of how state (context) may be preserved within an execution thread and across calls to **process_message()**, "A Complex Dequeuing Example".

The process_done() Routine

To assist in cleaning up state information for a thread, callers can provide a routine pointed to by the **process_done** call argument of mtaDequeueStart().

The following code example shows the required syntax for "The process_done() Routine".

```
void process_done(void *ctx2, void *ctx1)
```

Table 4–3 lists the arguments required for "The process_done() Routine", and gives a description of each.

Table 4–3 The Process_Done Routine Arguments

Required Arguments	Description
ctx2	The value of the last pointer stored by **process_message()** in the **ctx2** call argument for this thread.
ctx1	The caller-supplied private context passed as **ctx1** to mtaDequeueStart().

The following code example demonstrates the type of actions taken by "The process_done() Routine".

```
void process_done(ctx2, ctx1)
{
    struct our_state_t *state = (struct our_state_t *)ctx2;
    if (!state)
         return;
    /*
     * Take steps to undo the state
     * (for example, close any sockets or files)
     */
    ...

    /*
     * Free the memory allocated by process_message()
     * to store the state
     */
    free(state)
}
```

A Complex Dequeuing Example

The program shown in "Example 4-2 A Complex Dequeue" is a more complicated version of the simple example (see "A Simple Dequeuing Example"). In this example, more than one concurrent dequeue thread is permitted. Additionally, better use is made of the context support provided by mtaDequeueStart(), and a procedure to clean up and dispose of per-thread contexts is provided.

After the Messaging Server product is installed, these programs can be found in the following location:

MessagingServer_home/**examples/mtasdk/**

Some lines of code are immediately preceded by a comment of the format:

```
/* See explanatory comment N */
```

where *N* is a number. The numbers are links to some corresponding explanatory text in the section that follows this code, see Explanatory Text for Numbered Comments in the Complex Dequeue Example.

For the output generated by this code, see "Output from the Complex Dequeue Example".

Example 4-2 A Complex Dequeue

```
/*
 * dequeue_complex.c
 *
 * Dequeuing with more than one thread used.
 *
 */
#include &lt;stdio.h>
#include &lt;stdlib.h>
#if !defined(_WIN32)
#include &lt;unistd.h>
#endif
#include &lt;string.h>
#include &lt;sys/types.h>
#include &lt;sys/stat.h>
#include &lt;fcntl.h>
#include &lt;errno.h>
#include "mtasdk.h"

/* See explanatory comment 1 */
typedef struct {
    int debug;    /* Debug flag                                    */
    int max_count;/* Maximum. number of messages per BSMTP file */
} my_global_context_t;

/* See explanatory comment 2 */
typedef struct {
     int   id;         /* Dequeue threads id                    */
     FILE *fp;         /* Dequeue threads current output file   */
     int   count;      /* Messages output by this dequeue thread */
} my_thread_context_t;

static const char *NotifyToStr(int ret_type, char *buf);
static const char *UniqueName(char *buf, size_t maxbuf,
                              const char *suffix);
static mta_dq_process_done_t process_done;
static mta_dq_process_message_t process_message;

int main()
{
     my_global_context_t gctx;
     int ires;

     /*
      * Initialize the MTA SDK
      */
```

```
    if ((ires = mtaInit(0)))
    {
        mtaLog(mtaInit() returned %d; %s\n, ires,
               mtaStrError(ires, 0));
        return(1);
    }

    /*
     *  The global context is shared by all dequeue threads
     *  calling process_message() as a result of a given call
     *  to mtaDequeueStart().  The global context in this
     *  example provides process_message() with the following:
     *    (1) How many messages to put into a BSMTP file before
     *        closing it and starting a new one, and
     *    (2) Whether or not to produce diagnostic debug output.
     */
    /* See explanatory comment 3 */
    gctx.debug     = 1;
    gctx.max_count = 5;

    /*  Start the dequeue loop */
    /* See explanatory comment 4 */
    ires = mtaDequeueStart((void *)&amp;gctx, process_message,
                           process_done, 0);

    /* Check the return status */
    /* See explanatory comment 5 */
    if (!ires)
        /* Success */
        return(0);

    /* Produce an error message */
    /* See explanatory comment 6 */
    mtaLog("mtaDequeueStart() returned %d; %s", ires,
           mtaStrError(ires, 0));
    /* And exit with an error */
    returnh(1);
}

/* process_done() -- Called by mtaDequeueStart() to clean up
 * and destroy a per-thread context created by process_message().
 * See explanatory comment 7
 */
static void process_done(void *my_ctx_2, void *my_ctx_1)
{
    my_global_context_t *gctx = (my_global_context_t *)my_ctx_1;
    my_thread_context_t *tctx = (my_thread_context_t *)my_ctx_2;
    if (!tctx)
        return;

    /* Generate any requested diagnostic output requested? */
    /* See explanatory comment 8 */
    if (gctx &amp;&amp; gctx->debug)
        mtaLog("Dequeue thread done: id=%d; context=%p; "
               "messages=%d", tctx->id, tctx, tctx->count);

    /* Now clean up and destroy the context */
    if (tctx->fp)
    {
        fprintf(tctx->fp, "QUIT\n");
```

```
          fclose(tctx->fp);
     }
     free(tctx);
}

/*
 * process_message() -- Called by mtaDequeueStart() to process a
 *                      single message.
 * See explanatory comment 9
 */
static int process_message(void **my_ctx_2, void *my_ctx_1,
                           mta_dq_t *dq, const char *env_from,
                           size_t env_from_len)
{
     my_global_context_t *gctx;
     my_thread_context_t *tctx;
     int ires, ret_type;
     const char *to, *env_id, *line;
     size_t len;
     char notify_buf[100];

     /* This should never happen, but just to be safe we check */
     if (!my_ctx_1 || !my_ctx_2)
          return(MTA_ABORT);

     /* The pointer to our global context was passed as my_ctx_1 */
     /* See explanatory comment 10 */
     gctx = (my_global_context_t *)my_ctx_1;

     /*
      *  In this example, we just use the per-thread context to:
      *  (1) Track the output file for this dequeue thread across
      *      repeated calls, and
      *  (2) to count how many messages have been output by this
      *      dequeue thread.
      * See explanatory comment 11
      */
     if (!(*my_ctx_2))
     {
         /* First call to process_message() by this dequeue thread.
          * Store a pointer to our context.
          */
         tctx = (my_thread_context_t *)
                 calloc(1, sizeof(my_thread_context_t));
         if (!tctx)
              /* Insufficient virtual memory; give up now */
              return(MTA_ABORT);
         *my_ctx_2 = (void *)tctx;

         /* Debug output? */
         if (gctx->debug)
         {
              tctx->id = mtaDequeueThreadId(dq);
              mtaLog("Dequeue thread starting: id=%d; context=%p",
                      tctx->id, tctx);
         }
     }
     else
         /*
          *  This dequeue thread has already called
```

```
     *  process_message() previously.
     */
    tctx = (my_thread_context_t *)(*my_ctx_2);

/* Send a HELO or a RSET? */
if (0 == (tctx->count % gctx->max_count))
{
     char buf[1024];
     int fd;

     /* Need to send a HELO */

     /* Send a QUIT if weve already sent a HELO previously */
     if (tctx->count > 0 &amp;&amp; tctx->fp)
     {
        fprintf(tctx->fp, "QUIT\n");
        fclose(tctx->fp);
        tctx->fp = NULL;
     }

     /* Now open a file */
     fd = open(UniqueName(buf, sizeof(buf), ".bsmtp"),
               O_WRONLY | O_CREAT | O_EXCL, 0770);

     if (fd &lt; 0 || !(tctx->fp = fdopen(fd, "w")))
         return(MTA_ABORT);

     /* Now send the HELO */
     fprintf(tctx->fp, "HELO %s\n", mtaChannelToHost(NULL,
             NULL, MTA_DQ_CONTEXT, dq, 0));
 }
 else
 {
     /*
      *  Weve already sent a HELO. Send a RSET to start a new
      *  message.
      */
     fprintf(tctx->fp, "RSET\n");
 }
 tctx->count++;

 /*
  *  Output the command
  *     MAIL FROM: &lt;from-adr> RET=return-type ENVID=id
  */
 env_id = NULL;
 /* See explanatory comment 12 */
 ret_type = MTA_NOTIFY_DEFAULT;
 mtaDequeueInfo(dq, MTA_ENV_ID, &amp;env_id, NULL,
                MTA_NOTIFY_FLAGS, &amp;ret_type, 0);
 fprintf(tctx->fp, "MAIL FROM:&lt;%s> RET=%s%s%s\n", env_from,
        NotifyToStr(ret_type, NULL),
          (env_id ? " ENVID=" : ""),(env_id ? env_id : ""));
  /*  Output the command
   *    RCPT TO: &lt;to-adr> NOTIFY=notify-type
   *  for each recipient address
   *  See explanatory comment 13
   */
 while (!(ires =
           mtaDequeueRecipientNext(dq, &amp;to, &amp;len,
```

```
                                        MTA_NOTIFY_FLAGS, &amp;ret_type, 0)))
      {
           fprintf(tctx->fp, "RCPT TO:&lt;%s> NOTIFY=%s\n", to,
                  NotifyToStr(ret_type, notify_buf));

           /* Indicate that delivery to this recipient succeeded */
           /*  See explanatory comment 14 */
           mtaDequeueRecipientDisposition(dq, to, len,
                                        MTA_DISP_DELIVERED, 0);
     }
     /*
      *  If ires == MTA_EOF, then we exited the loop normally;
      *  otherwise, theres been an error of some sort.
      *  See explanatory comment 15
      */
     if (ires != MTA_EOF)
          return(ires);

     /* Now output the message itself */
     fprintf(tctx->fp, "DATA\n");
     /*  See explanatory comment 16 */
     while (!(ires = mtaDequeueLineNext(dq, &amp;line, &amp;len)))
     {
          /* Check to see if we need to dot-stuff the link */
          if (len == 1 &amp;&amp; line[0] == .)
          fprintf(tctx->fp, ".");

          /* Now output the line */
          /*  See explanatory comment 17 */
          fprintf(tctx->fp, "%.*s\n", len, line);
     }

     /*
      *  If ires == MTA_EOF, then we exited the loop normally;
      *  If ires == MTA_EOF, then we exited the loop normally;
      *  otherwise, theres been an error of some sort.
      */
     if (ires != MTA_EOF)
          return(ires);

     /* Output the "." command to terminate this message */
     fprintf(tctx->fp, ".\n");

     /* And dequeue the message */
     /*  See explanatory comment 18 */
     ires = mtaDequeueMessageFinish(dq, 0);

     /* All done; might as well return ires as our result */
     return(ires);
}

/*
 *  Convert a bitmask of MTA_NOTIFY_ flags to a readable string
 */
/*  See explanatory comment 19 */
static const char *
NotifyToStr(int ret_type, char *buf)
{
     if (!buf)
    /* Doing a RET= parameter to a MAIL FROM command */
```

```
        return((ret_type &amp; MTA_NOTIFY_CONTENT_FULL) ?
                "FULL" : "HDRS");
     buf[0] = \0;

     if (ret_type &amp; MTA_NOTIFY_SUCCESS)
          strcat(buf, "SUCCESS");

    if (ret_type &amp; MTA_NOTIFY_FAILURE)
    {
         if (buf[0])
              strcat(buf, ",");
         strcat(buf, "FAILURE");
    }
    if (ret_type &amp; MTA_NOTIFY_DELAY)
    {
        if (buf[0])
             strcat(buf, ",");
        strcat(buf, "DELAY");
     }

     if (!buf[0])
          strcat(buf, "NEVER");
     return(buf);
}
/* Generate a unique string suitable for use as a file name */
/*  See explanatory comment 20 */
static const char *
UniqueName(char *buf, size_t maxbuf, const char *suffix)
{
     strcpy(buf, "/tmp");
     mtaUniqueString(buf+5, NULL, maxbuf-5);
     strcat(buf, suffix);
     return(buf);

}
```

Explanatory Text for Numbered Comments in the Complex Dequeue Example

The numbered list that follows has explanatory text that corresponds to the numbered comments in "Example 4-2 A Complex Dequeue":

1. The global context data structure for this example. This is passed to mtaDequeueStart(), as the **ctx1** call argument.
2. Per-thread data structure used by dequeue threads. While mtaDequeueStart() creates each dequeue thread, it is up to the **process_message()** routine to actually create any per-thread context it might need.
3. Initialize the global context before calling mtaDequeueStart().
4. Initiate dequeue processing by calling **mtaDequeueStart()**. The first call argument is a pointer to the global context. Each time **mtaDequeueStart()** calls **process_message()**, it passes in the global context pointer as the second argument. In this example, mtaDequeueStart() is not told to limit the number of dequeue threads it uses.
5. If the call to mtaDequeueStart() succeeds, the program exits normally.
6. If the call to mtaDequeueStart() fails, then a diagnostic error message is displayed and the program exits with an error status.

7. Each dequeue thread calls **process_done()** as it exits. This program cleans up and destroys any per-thread contexts created by the **process_message()** routine.
8. The program generates optional diagnostic output. Calling **mtaLog()** directs the output to the appropriate location: **stdout** if the program is run manually, and the channel log file if the program is run by the Job Controller.
9. mtaDequeueStart() calls **process_message()** once for each queued message to be processed. On the first call, the memory pointed at by **my_ctx_2** is guaranteed to be **NULL**. The value of the first call argument passed to mtaDequeueStart() is passed to **process_message()** as the **my_ctx_1** call argument.
10. The global context contains information pertinent to all the dequeue threads generated by the call **mtaDequeueStart()**.
11. **process_message()** uses a per-thread context to save data across all calls to itself by a single dequeue thread.
12. **mtaDequeueInfo()** is used to obtain the envelope ID and RFC 1891 notification flags, if any, associated with the message being processed.
13. **mtaDequeueRecipientNext()** is used to obtain each envelope recipient address, one address per call. When there are no more recipient addresses to obtain, the routine returns the status **MTA_EOF**.
14. Each recipient is marked as delivered with a call to mtaDequeueRecipientDisposition(). An actual channel program would typically not make this call until after processing the message further.
15. If **process_message()** returns without dequeuing the message, mtaDequeueStart() defers the message for a later delivery attempt.
16. The message header and body are read one line at a time with **mtaDequeueLineNext()**. When there are no more lines to read, it returns a status of **MTA_EOF**.
17. Lines returned by **mtaDequeueLineNext()** might not be **NULL** terminated because the returned line pointer might point to a line in a read-only, memory-mapped file.
18. **mtaDequeueMessageFinish()** is called once the message had been fully processed and the disposition of all its recipients set with mtaDequeueRecipientDisposition(). The message is not truly dequeued until this happens.
19. The routine **NotifyToStr()** converts a bitmap encoded set of RFC 1891 notification flags to an ASCII text string.
20. The **UniqueName()** routine generates a unique string suitable for the use as a file name. This is used to generate the unique portion of the file name. This routine can be called concurrently by multiple threads and always generates a string unique amongst all processes and threads on the system.

For information on how to run this sample program, see "Running Your Enqueue and Dequeue Programs".

Output from the Complex Dequeue Example

The output that follows shows the result of 100 queued messages processed with the program in "Example 4-2 A Complex Dequeue".

```
11:01:16.82: Dequeue thread starting: id=10; context=32360
11:01:16.87: Dequeue thread starting: id=1; context=32390
11:01:16.93: Dequeue thread starting: id=2; context=325e8
```

```
11:01:17.00: Dequeue thread starting: id=3; context=32600
11:01:17.04: Dequeue thread starting: id=4; context=32618
11:01:17.09: Dequeue thread starting: id=5; context=32630
11:01:17.14: Dequeue thread starting: id=6; context=78e50
11:01:17.19: Dequeue thread starting: id=7; context=88a18
11:01:17.23: Dequeue thread starting: id=9; context=8ab78
11:01:17.51: Dequeue thread starting: id=8; context=8ab60
11:01:19.96: Dequeue thread done: id=2; context=325e8; messages=12
11:01:19.96: Dequeue thread done: id=5; context=32630; messages=22
11:01:19.97: Dequeue thread done: id=6; context=78e50; messages=11
11:01:19.97: Dequeue thread done: id=4; context=32618; messages=5
11:01:19.98: Dequeue thread done: id=8; context=8ab60; messages=16
11:01:20.00: Dequeue thread done: id=9; context=8ab78; messages=5
11:01:20.00: Dequeue thread done: id=3; context=32600; messages=12
11:01:20.01: Dequeue thread done: id=1; context=32390; messages=7
11:01:20.02: Dequeue thread done: id=10; context=32360; messages=6
11:01:20.03: Dequeue thread done: id=7; context=88a18; messages=4
```

Intermediate processing channels

Special attention is warranted for intermediate processing channels. Intermediate processing channels are channels which re-enqueue back to the MTA the mail they dequeue from it. For example, a virus scanner or a conversion channel, which, after scanning or converting a message, re-enqueues it back to the MTA for further routing or delivery. Such channels should do the following:

- Preserve Envelope Information
- Use MTA_ENV_TO
- Use Rewrite Rules to Prevent Message Loops

The sample code, "Intermediate Channel Example", illustrates the SDK usage required to effect the first two preceding points.

Preserve Envelope Information

All queued messages have envelope fields which are unique to the message. For instance, a message will have the RFC 1891 envelope ID that was either assigned by the MTA when the message was first enqueued, or was specified by a remote MTA and transmitted over SMTP. The same applies to the RFC 1891 original recipient address fields that specify the original form of each of the message's envelope recipient addresses. Furthermore, there may be other envelope fields which have non-default settings such as notification handling flags. Whenever possible, this information should be preserved as the message flows from MTA channel to MTA channel. In order to preserve this information, it must be copied from the message being dequeued to the new message being enqueued. This copying process is best done using the **MTA_DQ_CONTEXT** item code in conjunction with the "mtaDequeueStart()" and **mtaEnqueueTo()** routines. When used with the former, it causes per-message envelope information to be automatically copied from the message being dequeued to the new message being enqueued. When used with the latter, it causes per-recipient information to be automatically copied.

Channel programs should not attempt to explicitly copy envelope information other than the envelope **From:** and envelope recipient addresses. The **MTA_DQ_CONTEXT** item code should always be used to implicitly perform the copy. The reason for this is straightforward: if a program attempts to do the copy explicitly by querying the fields one by one from the message being dequeued, and then setting them one by one in the message being enqueued, then any new envelope fields introduced in later versions of

Messaging Server will be lost unless the program is updated to explicitly know about those new fields too.

Use MTA_ENV_TO

Intermediate processing channels should use the **MTA_ENV_TO** item code with **mtaEnqueueTo()** rather than the **MTA_TO**, **MTA_CC**, and **MTA_BCC** item codes. This tells the MTA that the recipient address being specified should be added to only the message's envelope and not also to a **Resent-To:**, **Resent-Cc:**, or **Resent-Bcc:** header line. "Example 4-3 Intermediate Channel Example", and illustrate the use of the **MTA_ENV_TO** item code. Both of those examples represent intermediate processing channels which are handling a previously constructed message. As such, they do not need to alter the existing message header.

Use Rewrite Rules to Prevent Message Loops

Finally, intermediate processing channels often require special rewrite rules in order to prevent message loops. Specifically, loops in which mail re-enqueued by the intermediate processing channel is queued back to the intermediate processing channel. See "Preventing Mail Loops when Re-enqueuing Mail" for further information on this topic.

Intermediate Channel Example

The sample program in this section, in "Example 4-3 Intermediate Channel Example", converts the body of each queued message and then re-enqueues the converted messages back to the MTA. The conversion process involves applying the rot 13 encoding used by some news readers to encode potentially offensive message content.

To configure the MTA to run this channel, see "Running Your Enqueue and Dequeue Programs". Also refer to "Preventing Mail Loops when Re-enqueuing Mail", which discusses configuring special rewrite rules for programs re-enqueuing dequeued email.

Some lines of code in this example are immediately preceded by a comment of the format:

```
/* See explanatory comment N */
```

where *N* is a number.

The numbers are links to some corresponding explanatory text found in "Explanatory Text for Numbered Comments in the Intermediate Channel Example".

Example 4-3 Intermediate Channel Example

```
/* intermediate_channel.c
 * A channel program that re-enqueues queued messages after first
 * transforming their content with the "rot13" transformation.
 */
#include &lt;stdio.h>
#include &lt;stdlib.h>
#include "mtasdk.h"

typedef struct {
     size_t  maxlen;
     char   *buf;
} rot13_buf_t;

static mta_dq_process_done_t process_done;
```

```
static mta_dq_process_message_t process_message;
static char rot13(char c);
static const char *rot13str(rot13_buf_t **dst, const char *src,
                            size_t srclen);

int main()
{
     int ires;

     /*
      * Initialize the MTA SDK
      */
     if ((ires = mtaInit(0)))
     {
         mtaLog(mtaInit() returned %d; %s\n, ires,
                mtaStrError(ires, 0));
         return(1);
     }

     /*
      *  Start the dequeue loop
      *  See explanatory comment 1
      */
     ires = mtaDequeueStart(NULL, process_message,
                            process_done, 0);

      /*
       *  Check the return status
       * See explanatory comment 2
       */
      if (!ires)
           /*
            *  Success
            */
           return(0);
      /*
       *  Produce an error message
       * See explanatory comment 3 */
       */
      mtaLog("mtaDequeueStart() returned %d; %s", ires,
              mtaStrError(ires, 0));

      /*
       *  And exit with an error
       */
      return(1);
}

/*
 *  process_done -- Clean up the private context my_ctx_2 used by
 *                  process_message.
 * See explanatory comment 4
 */
static void process_done(void *my_ctx_2, void *my_ctx_1)
{
     rot13_buf_t *rbuf;

     if (!my_ctx_2)
          return;
     rbuf = (rot13_buf_t *)my_ctx_2;
```

```
     if (rbuf->buf)
          free(rbuf->buf);
     free(rbuf);
}


/*
 *  process_message -- Process a single message by re-enqueuing but
 *                     with its message body converted to the rot13
 *                     set. The private my_ctx_1 context is not
 *                     used. The private my_ctx_2 context is used
 *                     for a rot13 translation context.
 * See explanatory comment 5
 */

static int process_message(void **my_ctx_2, void *my_ctx_1,
                           mta_dq_t *dq,
{
     size_t len;
     const char *line, *to;
     int in_header;
     mta_nq_t *nq;

     /*
      *  Start a message enqueue
      */
     nq = NULL;
     /* See explanatory comment 6 */
     if (mtaEnqueueStart(&amp;nq, env_from, env_from_len,
          MTA_DQ_CONTEXT, dq, 0))
             goto(defer);

     /*
      *  Process the envelope recipient list
      *  See explanatory comment 7 */
      */
     while (!mtaDequeueRecipientNext(dq, &amp;to, &amp;len, 0))
          /* See explanatory comment 7 */
          if (mtaEnqueueTo(nq, to, len, MTA_DQ_CONTEXT, dq,
              MTA_ENV_TO, 0) ||
              /* See explanatory comment 8 */
              mtaDequeueRecipientDisposition(dq, to, len,
                                             MTA_DISP_DELIVERED,0))
               /* See explanatory comment 9 */
               goto defer;
      if (mta_errno != MTA_EOF)
           goto defer;

      /*
       *  First, get the messages header and write it
       *  unchanged to the new message being enqueued.
       * See explanatory comment 10
       */
      in_header = 1;
      while (in_header &amp;&amp; !mtaDequeueLineNext(dq, &amp;line, &amp;len))
      {
           if (mtaEnqueueWriteLine(nq, line, len, 0))
                goto defer;
           if (!len)
                in_header = 0;
```

```
    }

    /*
     *  Determine why we exited the while loop
     */
    if (in_header)
    {
        /*
         *  We exited before seeing the body of the message
         * See explanatory comment 12
         */
        if (mta_errno == MTA_EOF)
             /*
              *  Message read completely: it must have no body
              */
             goto done;
        else
             /*
              *  Error condition of some sort
              */
             goto defer;
     }

     /*
      *  Now rot13 the body of the message
      * See explanatory comment 13
      */
     while (!mtaDequeueLineNext(dq, &amp;line, &amp;len))
           if (mtaEnqueueWriteLine(nq,
                           rot13str((rot13_buf_t **)my_ctx_2,
                           line, len), len, 0))
                goto defer;

     /*
      * If mta_errno == MTA_EOF, then we exited the loop
      * normally; otherwise, theres been an error of some sort
      */
     if (mta_errno != MTA_EOF)
          goto defer;

     /*
      *  All done, enqueue the new message
      * See explanatory comment 14
      */
done:
     if (!mtaEnqueueFinish(nq, 0) &amp;&amp;
         !mtaDequeueMessageFinish(dq, 0))
          return(0);
     /*
      *  Fall through to defer the message
      */
     nq = NULL;

     /*
      *  A processing error of some sort has occurred: defer the
      *  message for a later delivery attempt
      * See explanatory comment 15
      */
defer:
     mtaDequeueMessageFinish(dq, MTA_ABORT, 0);
```

```
       if (nq)
            mtaEnqueueFinish(nq, MTA_ABORT, 0);
       return(MTA_NO);
}

/*
 *  rot13 -- an implmentation of the rotate-by-13 translation
 * See explanatory comment 16
 */
static char rot13(char c)
{
     if (A &lt;= c &amp;&amp; c &lt;= Z)
        return (((c - A + 13) % 26) + A);
     else if (a &lt;= c &amp;&amp; c &lt;= z)
        return (((c - a + 13) % 26) + a);
     else return (c);
}

/*
 *  rot13str -- Perform a rot13 translation on a string of text
 *  See explanatory comment 17
 */
static const char *rot13str(rot13_buf_t **dst, const char *src,
                            size_t srclen)
{
     size_t i;
     char *ptr;
     rot13_buf_t *rbuf = *dst;

     /*
      *  First call?  If so, then allocate a rot13_buf_t structure
      */
     if (!rbuf)
     {
          rbuf = calloc(1, sizeof(rot13_buf_t));
          if (!rbuf)
               return(NULL);
          *dst = rbuf;
     }

     /*
      *  Need a larger buffer?
      *  If so, then increase the length of rbuf->buf
      */
     if (rbuf->maxlen &lt; srclen || !rbuf->buf)
     {
          size_t l;
          char *tmp;
          /* Round size up to the nearest 2k */
          l = 2048 * (int)((srclen + 2047) / 2048);
          tmp = (char *)malloc(l);
          if (!tmp)
               return(NULL);
          if (rbuf->buf)
               free(rbuf->buf);
          rbuf->buf    = tmp;
          rbuf->maxlen = l;
      }
      /*
       *  Now rot13 our input
```

```
         */
        ptr = rbuf->buf;
        for (i = 0; i &lt; srclen; i++)
             *ptr++ = rot13(*src++);

        /*
         *  All done
         */
        return(rbuf->buf);
}
```

Explanatory Text for Numbered Comments in the Intermediate Channel Example

1. The dequeue processing is initiated by calling mtaDequeueStart(). In this example, no global context is used; hence, the first call argument to **mtaDequeueStart()** is **NULL**.
2. If the call to mtaDequeueStart() succeeds, then the program exits normally.
3. If the call to mtaDequeueStart() fails, a diagnostic error message is displayed and the program exits with an error status.
4. Each dequeue thread calls "The process_done() Routine" as it exits. The intent is to allow the program to clean up and destroy any per-thread contexts created by the **process_message()** routine. In this case, the buffer used by **rot13str()** is deallocated.
5. The mtaDequeueStart() routine calls **process_message()** once for each queued message to be processed. On the first call by a dequeue thread, the memory pointed at by **my_ctx_2** is **NULL**.
6. A message enqueue starts. The dequeue context, **dq**, is provided so that per-message envelope fields can be carried over to the new message from the message being dequeued.
7. Each envelope recipient address is obtained, one at a time, with **mtaDequeueRecipientNext()**. When there are no more recipient addresses to obtain, **mtaDequeueRecipientNext()** returns the status **MTA_EOF**.
8. Each envelope recipient address is added to the recipient list for the new message being enqueued. The **MTA_ENV_TO** option for **mtaEnqueueTo()** is specified so that the address is to be added to the new message's envelope only. It should not also be added to the message's RFC 822 header. The new message's header will be a copy of the header of the message being dequeued. This copy is performed at the code location marked by comment 12.
9. Each recipient is marked as delivered with mtaDequeueRecipientDisposition().
10. In the event of an error returned from either **mtaEnqueueTo()** or mtaDequeueRecipientDisposition(), or an unexpected error return from **mtaDequeueRecipientNext()**, the ongoing enqueue is cancelled and the processing of the current message is deferred.
11. Each line of the current message is read and then copied to the new message being enqueued. This copying continues until a blank line is read from the current message. (A blank line signifies the end of the RFC 822 message header and the start of the RFC 822 message content.)
12. The code here needs to determine why it exited the read loop: because of an error, or because the transition from the message's header to body was detected.

13. The remainder of the current message is read line by line and copied to the new message being enqueued. However, the line enqueued is first transformed using the rot13 transformation. The per-thread context **my_ctx_2** is used to hold an output buffer used by the **rot13str()** routine.
14. The enqueue of the new message is finished. If that step succeeds, then the message being dequeued is removed from the MTA queues.
15. In the event of an error, the new message enqueue is cancelled and the current message left in the queues for later processing.
16. The rot13 character transformation.
17. A routine that applies the rot13 transformation to a character string.

Sample Input Message for the Intermediate Channel Example

The example that follows is a sample input message from the queue to be processed by the program found in "Example 4-3 Intermediate Channel Example".

```
Received: from frodo.west.siroe.com by frodo.west.siroe.com
 (Sun Java System Messaging Server 6 2004Q2(built Mar 24 2004))id
&lt;0HCH00301E6GO700@frodo.west.siroe.com\> for sue@sesta.com; Fri,
 28 Mar 2003 14:51:52 -0800 (PST)
Date: Fri, 28 Mar 2003 14:51:52 -0800 (PST)
From: root@frodo.west.siroe.com
Subject: Testing
To: sue@sesta.com
Message-id: &lt;0HCH00303E6GO700@frodo.west.siroe.com\>
MIME-version: 1.0

This is a test message.
```

Output from the Intermediate Channel Example

This example shows the output generated by the dequeue and re-enqueue program.

```
Received: from sesta.com by frodo.west.siroe.com
 (Sun Java System Messaging Server 6 2004Q2 (built Mar 24 2003))id
&lt;0HCH00301E7DOH00@frodo.west.wiroe.com\> for sue@sesta.com; Fri,
 28 Mar 2003 14:51:58 -0800 (PST)
Received: from frodo.west.siroe.com by frodo.west.siroe.com
 (Sun Java System Messaging Server 6 2004Q2 (built Mar 24 2003))id
&lt;0HCH00301E7DOH00@frodo.west.wiroe.com\> for sue@sesta.com; Fri,
 28 Mar 2003 14:51:52 -0800 (PST)
Date: Fri, 28 Mar 2003 14:51:52 -0800 (PST)
From: root@frodo.west.siroe.com
Subject: Testing
To: sue@sesta.com
Message-id: &lt;0HCH00303E6GO700@frodo.west.siroe.com\>
MIME-version: 1.0

Guvf vf n grfg zrffntr.
```

Thread Creation Loop in mtaDequeueStart()

After mtaDequeueStart() performs any necessary initialization steps, it then starts a loop whereby it communicates with the MTA Job Controller. Based upon information from the Job Controller, it then creates zero or more execution threads to process queued messages.

While any execution threads are running, the thread that invoked mtaDequeueStart()(the primal thread) executes a loop containing a brief pause (that is, a sleep request). Each time the primal thread awakens, it communicates with the Job Controller to see if it should create more execution threads. In addition, the Job Controller itself has logic to determine if more threads are needed in the currently running channel program, or if it should create additional processes to run the same channel program.

To demonstrate, the following code example shows pseudo-code of the mtaDequeueStart() loop.

```
threads_running = 0
threads_max = MTA_THREAD_MAX_THREADS
attemtps    = MTA_JBC_MAX_ATTEMPTS

LOOP:
    while (threads_running &lt; threads_max)
    {

      Go to DONE if a shut down has been requested

      pending_messages = Ask the Job Controller how many
                         messsages there are to be processed

      // If there are no pending messages
      // then consider what to do next
      if (pending_messages = 0)
      {
          // Continue to wait?
          if (attempts &lt;= 0)
              go to DONE

          // Decrement attempts and wait
          attempts = attempts - 1;
          go to SLEEP
      }
      // Reset the attempts counter
      attempts = MTA_JBC_MAX_ATTEMPTS

      threads_needed = Ask the Job Controller how many
                       processing threads are needed

      // Cannot run more then threads_max threads per process
      if (threads_needed \> threads_max)
          threads_needed = threads_max

      // Create additional threads if needed
      if (threads_needed \> threads_running)
      {
         Create (threads_needed - threads_running) more threads
         threads_running = threads_needed
      }
    }

SLEEP:
    Sleep for MTA_JBC_RETRY_INTERVAL seconds
     -- a shut down request will cancel the sleep
    go to LOOP

DONE:
```

```
Wait up to MTA_THREAD_WAIT_TIMEOUT seconds
for all processing threads to exit

Return to the caller of mtaDequeueStart()
```

Multiple Calls to mtaDequeueStart()

A channel program can call mtaDequeueStart() multiple times, either sequentially or in parallel. In the latter case, the program would need to create threads so as to effect multiple, simultaneous calls to mtaDequeueStart(). However, just because this can be done does not mean that it is appropriate to do so. In the former case of multiple sequential calls, there is no need to be making repeated calls. When mtaDequeueStart() returns, the channel no longer needs immediate processing and has been in that state for the number of seconds represented by the following formula:

MTA_JBC_ATTEMPTS_MAX * MTA_JBC_RETRY_INTERVAL

Instead, the channel program should exit thereby freeing up system resources. The Job Controller will start a new channel program running when there are more messages to process.

In the latter case of multiple parallel calls, there is again no need to do so. If there is an advantage to running more threads than a single call generates, then the channel's **threaddepth** channel option setting should be increased so that a single call does generate more threads.

The only exception to either of these cases might be if the multiple calls are each for a different channel. Even then, however, the advantage of so doing is dubious as the same effect can be achieved through the use of multiple processes, one for each channel.

Calling Order Dependencies

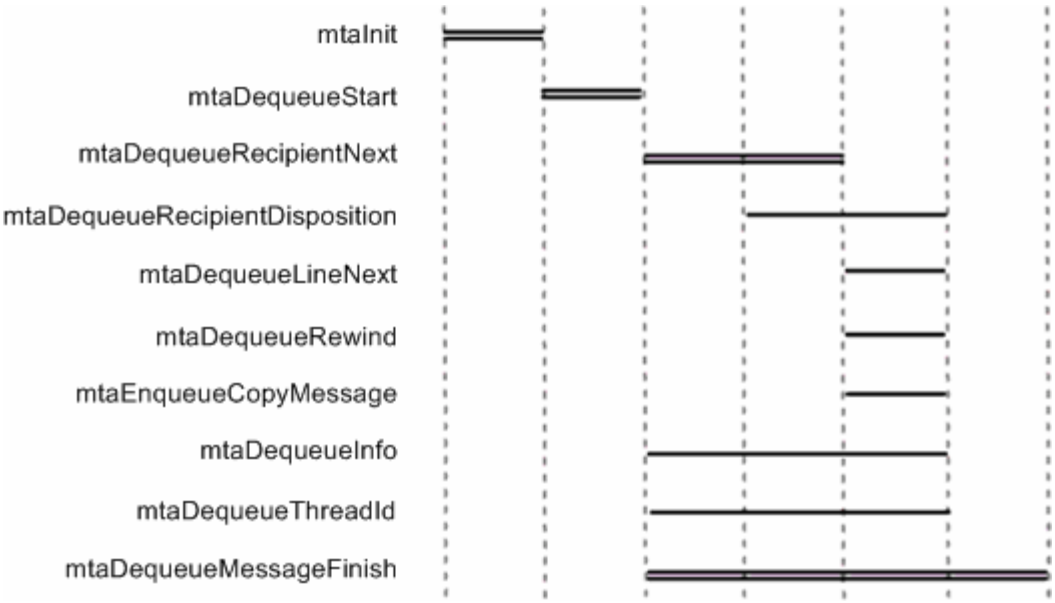
When you are constructing programs, there is a calling order for the MTA SDK routines that must be observed; some routines must be called before others.

"Example 4-1 A Simple Dequeue" visually depicts the calling order dependency of the message dequeue routines. To the right of each routine name appears a horizontal line segment, possibly broken across a column, for example, **mtaDequeueRecipientNext()**. Routines for which two horizontal line segments, one atop the other, appear are required routines; that is, routines that must be called in order to successfully enqueue a message. The required routines are **mtaInit()**, mtaDequeueStart(), **mtaDequeueRecipientNext()**, and **mtaDqueueMessageFinish()**.

To determine at which point a routine may be called, start in the leftmost column and work towards the rightmost column. Any routine whose line segment lies in the first (leftmost) column may be called first. Any routine whose line segment falls in the second column may next be called, after which any routine whose line segment falls in the third column may be called, and so forth. When more than one routine appears in the same column, any or all of those routines may be called in any order. Progression from left to right across the columns is mandated by the need to call the required routines.

After calling mtaDequeueRewind(), the read point into the underlying queued message file is reset to the start of the message's outermost header; that is, you are back in the third column. Figure 4–1 shows the calling order dependency for message dequeue routines.

Figure 4–1 Calling Order Dependency for Message Dequeue Routines



5

Decoding Messages

The MTA has facilities for parsing and decoding single and multipart messages formatted using the MIME Internet messaging format. Additionally, these facilities can convert messages with other formats to MIME. For example, messages with BINHEX or UUENCODE data, the RFC 1154 format, and many other proprietary formats. The mtaDecodeMessage() routine provides access to these facilities, parsing either a queued message or a message from an arbitrary source such as a disk file or a data stream.

Usage Modes for mtaDecodeMessage()

There are two usage modes for mtaDecodeMessage(). In the first mode, messages are simply parsed, any encoded content decoded, and each resulting, atomic message part presented to an inspection routine. This mode of usage is primarily of use to channels which interface the MTA to non-Internet mail systems such as SMS and X.400. The second mode of operation allows the message to be rewritten after inspection. The output destination for this rewriting may be either the MTA channel queues, or an arbitrary destination via a caller-supplied output routine. During the inspection process in this second usage mode, individual, atomic message parts may be discarded or replaced with text. This operational mode is primarily of use to intermediate processing channels which need to scan message content or perform content conversions. For example, virus scanners and encryption software. "A Simple Decoding Example" illustrates the first usage mode, while "A Simple Virus Scanner Example" the second.

For the first usage mode, the calling routine must supply the following items:

1. An input source for the message.
2. An inspection routine which will be passed each atomic message part of the parsed and decoded message. For the second usage mode, the calling routine must supply the same two items as listed for the first usage mode, and in addition a third item must be supplied:
3. An output destination to direct the resulting message to.

The input source can be either a queued message file, represented by a dequeue context, or it can be provided by a caller-supplied input routine. Use the former when processing queued messages and the latter when processing data from disk files, data streams, or other arbitrary input sources. Since the parser and decoder require only a single, sequential pass over its input data, it is possible to stream data to mtaDecodeMessage().

The output destination can be a message being enqueued and represented either by an enqueue context, or by a caller-supplied output routine. Use an enqueue context when

submitting the message to the MTA. In all other cases, use a caller-supplied output routine.

The following are some common usage cases and their associated input sources and output destinations.

- **Send to the MTA (slave channel).** For this case, a caller- supplied routine accepts incoming messages from a source outside of the MTA and then enqueues it to the MTA. The caller-supplied input routine is used in conjunction with an enqueue context as the output source. Doing a MIME parse and decode is not usually called for in this case. However, specialized services might be constructed this way. For instance, a custom server that accepts MIME formatted messages, and strips a control attachment before submitting the remainder of the message to the MTA.
- **An intermediate processing channel.** For this case, an example is a virus scanner that scans queued mail messages, re-enqueuing them to the MTA for delivery. In this case, a dequeue context is used as the input source and an enqueue context as the output source.
- **Send from the MTA (master channel).** For this case, queued messages are gatewayed to another mail system. A dequeue context is used for the input source and an output destination is often not needed; the inspection routine usually suffices. Channels of this sort are common place when interfacing Messaging Server to systems that do not support MIME and for which conversion of MIME formatted messages to other formats is required (for example, X.400 and SMS).
- **A command line utility to parse a message.** For this case, a caller-supplied input routine is used. No output destination is needed; an inspection routine usually suffices.

The Input Source

The message to be decoded is provided as either a dequeue context or a caller-supplied routine.

Dequeue Context

When using a dequeue context, you must observe the following:

1. Pass the dequeue context from **mtaDecodeStart()** to mtaDecodeMessage() along with the **MTA_DECODE_DQ** item code.
2. The recipient list of the message being dequeued must have already been read by mtaDequeueRecipientNext() before calling mtaDecodeMessage().
3. **mtaDequeueMessageFinish()** must not yet have been called for the dequeue context.

After using a dequeue context with mtaDecodeMessage(), further calls to mtaDequeueRecipientNext() can't be made. Calls to mtaDequeueLineNext() can only be performed after a call to mtaDequeueRewind().

Caller-Supplied Input Routine

To use a caller-supplied input routine, pass the address of the input routine along with the **MTA_DECODE_PROC** item code to mtaDecodeMessage(). In "Example 5-1 Decoding MIME Messages Simple Example", the caller supplied routine's name is **decode_read()**.

When using a caller-supplied input routine, each block of data returned by the routine must be a single line of the message. This is the default expectation of

mtaDecodeMessage() and corresponds to the **MTA_TERM_NONE** item code. If, instead, the **MTA_TERM_CR**, **_CRLF**, **_LF**, or **_LFCR** item code are specified, then the block of data need not correspond to a single, complete line of message data; it may be a portion of a line, multiple lines, or even the entire message.

On each successful call, the input routine should return a status code of zero (**MTA_OK**). When there is no more message data to provide, then the input routine should return **MTA_EOF**. The call that returns the last byte of data should return zero; it is the subsequent call that must return **MTA_EOF**. In the event of an error, the input routine should return a non-zero status code other than **MTA_EOF** (for example, **MTA_NO**). This terminates the message parsing process and mtaDecodeMessage() returns an error.

The Inspection Routine

Whenever mtaDecodeMessage() is called, an inspection routine must be supplied by the caller. In "Example 5-1 Decoding MIME Messages Simple Example", the inspection routine's name is **decode_inspect()**.

As the message is parsed and decoded, mtaDecodeMessage() presents each atomic message part to the inspection routine one line at a time. The presentation begins with the part's header lines. Once all of the header lines have been presented, the lines of content are presented.

So that the inspection routine can tell if it is being presented with a line from the header or content of the message, a data type indicator is supplied to the inspection routine each time it is called. In regards to lines of the message's content, the data type indicator discriminates between text and binary content. Text content is considered any content with a MIME content type of **text** or **message** (for example, **text/plain**, **text/html**, **message/rfc822**), while binary content is all other MIME content types (**application**, **image**, and **audio**).

When writing an inspection routine for use with mtaDecodeMessage(), the following points apply:

- Message parts need not have any content. A common case is a single part message with no content for which the sender used the **Subject:** header line to express their communique.
- In the case of a non-multipart message, the message has a single part. The header for this sole part is the header for the message itself. As noted previously, there may or may not be any content to this single part.
- In the case of a multipart message, individual parts need not have a part header. In such cases, MIME's defaults apply and imply that the content is **text/plain** using the US-ASCII character set.
- Regardless of the value of the **Content-transfer-encoding** header line, the content presented will no longer be encoded.
- In the case of a multipart message, the outermost header is not presented. However, it may be inspected by means of an output routine (see "The Output Destination").

A Simple Decoding Example

This sample program found in "Example 5-1 Decoding MIME Messages Simple Example" decodes a MIME formatted message using mtaDecodeMessage(). This is not

a channel program. The actual message to be decoded is compiled into the program rather than being drawn from a channel queue.

After the Messaging Server product is installed, these programs can be found in the following location:

MessagingServer_home/**examples/mtasdk/**

Some lines of code are immediately preceded by a comment of the format:

```
/* See explanatory comment N */
```

where *N* is a number. The numbers are links to some corresponding explanatory text in the section that follows this code, see "Explanatory Text for Numbered Comments in the Simple Decoding Example".

For the sample output generated by this program, see "MIME Message Decoding Simple Example Output".

Example 5-1 Decoding MIME Messages Simple Example

```
/*
 *  decode_simple.c
 *
 *    Decode a multipart MIME message.
 *
 */
#include &lt;stdio.h>
#include &lt;string.h>
#include "mtasdk.h"

/*
 *  Inline data for a sample message to decode
 * See explanatory comment 1
 */
static const char message[] =
  "From: sue@siroe.com\n"
  "Date: 31 Mar 2003 09:32:47 -0800\n"
  "Subject: test message\n"
  "Content-type: multipart/mixed; boundary=BoundaryMarker\n"
  "\n\n"
  "--BoundaryMarker\n"
  "Content-type: text/plain; charset=us-ascii\n"
  "Content-disposition: inline\n"
  "\n"
  "This is a\n"
  "  test message!\n"
  "--BoundaryMarker\n"
  "Content-type: application/postscript\n"
  "Content-disposition: attachment; filename='a.ps'\n"
  "Content-transfer-encoding: base64\n"
  "\n"
  "IyFQUwoxMDAgMTAwIG1vdmV0byAzMDAgMzAwIGxpbmV0byBzdHJva2UKc2hv"  "3Bh\n"
  "Z2UK\n"
  "--BoundaryMarker--\n";

static mta_decode_read_t decode_read;
static mta_decode_inspect_t decode_inspect;
typedef struct {
     const char *cur_position;
     const char *end_position;
} position_t;
```

```
main()
{
     position_t pos;

     /*
      * Initialize the MTA SDK
      */
     if ((ires = mtaInit(0)))
     {
         mtaLog("mtaInit() returned %d; %s\n", ires,
                mtaStrError(ires, 0));
         return(1);
     }

     /*
      *  For a context to pass to mtaDecodeMessage(), we pass a
      *  pointer to the message data to be parsed.  The
      *  decode_read() routine uses this information when
      *  supplying data to mtaDecodeMessage().
      *  See explanatory comment 2
      */
     pos.cur_position = message;
     pos.end_position = message + strlen(message);

     /*
      *  Invoke mtaDecodeMessage():
      *    1. Use decode_read() as the input routine to supply the
      *       message to be MIME decoded,
      *    2. Use decode_inspect() as the routine to inspect each
      *       MIME decoded message part,
      *    3. Do not specify an output routine to write the
      *       resulting, MIME message, and
      *    4. Indicate that the input message source uses LF
      *       record terminators.
      *  See explanatory comment 3
      */
     mtaDecodeMessage((void *)&amp;amp;pos, MTA_DECODE_PROC,
                      (void *)decode_read,
                      0, NULL, decode_inspect, MTA_TERM_LF, 0);
}

/*
 *  decode_read -- Provide message data to mtaDecodeMessage().
 *                 The entire message could just as easily be
 *                 given to mtaDecodeMessage()at once. However,
 *                 for illustration purposes, the message is
 *                 provided in 200 byte chunks.
 *  See explanatory comment 4
 */
static int decode_read(void *ctx, const char **line, size_t
                       *line_len)
{
     position_t *pos = (position_t *)ctx;

     if (!pos)
          return(MTA_NO);
     else if (pos->cur_position >= pos->end_position)
          return(MTA_EOF);
     *line = pos->cur_position;
     *line_len = ((pos->cur_position + 200) &lt;
```

```
                pos->end_position) ? 200 :
                (pos->end_position - pos->cur_position);
     pos->cur_position += *line_len;
     return(MTA_OK);
}

/*
 *  decode_inspect -- Called by mtaDecodeMessage() to output a
 *                    a line of the parsed message.  The line is
 *                    simply output with additional information
 *                    indicating whether the line comes from a
 *                    header, text part, or binary part.
 *  See explanatory comment 5
*/
static int decode_inspect (void *ctx, mta_decode_t *dctx, int
                           data_type, const char *data,
                           size_t data_len)
{
     static const char *types[] = {"N", "H", "T", "B"};

     /* See explanatory comment 6 */
     if (data_type == MTA_DATA_NONE)
          return(MTA_OK);

     /* See explanatory comment 7 */
     printf("%d%s: %.*s\n",
            mtaDecodeMessageInfoInt(dctx,
                                    MTA_DECODE_PART_NUMBER),
                                    types[data_type], data_len,
                                    data);

            return(MTA_OK);
}
```

Explanatory Text for Numbered Comments in the Simple Decoding Example

The following numbered explanatory text corresponds to the numbered comments in "Example 5-1 Decoding MIME Messages Simple Example".

1. The MIME message to be decoded. It is a multipart message with two parts. The first part contains text, the second part a PostScript attachment.
2. The private context to be passed to mtaDecodeMessage()and, in turn, passed by it to the supplied input routine, **decode_read()**. The input routine uses this context to track how many bytes of the input message it has supplied to mtaDecodeMessage().
3. The call to mtaDecodeMessage(). An input routine, **decode_read()**, is supplied to provide the message to be decoded. Since the message source has each record terminated by line feeds, the **MTA_TERM_LF** option is also specified. The routine **decode_inspect()** is passed for use as an inspection routine.
4. The input routine, **decode_read()**. This routine provides the message to be decoded 200 bytes at a time. Note that providing only 200 bytes at a time is arbitrary: the routine could, if it chose, provide the entire message, or 2000 bytes at a time, or a random number of bytes on each call. After the entire message has been supplied, subsequent calls to **decode_read()** return the **MTA_EOF** status.

5. The inspection routine, **decode_inspect()**. For each atomic message part, this routine is called repeatedly. The repeated calls provide, line by line, the part's header and decoded content.
6. For a given message part, the final call to **decode_inspect()** provides no part data. This final call serves to give **decode_inspect()** a last chance to accept or discard the part when outputting the final form of the message via an optional output routine supplied to mtaDecodeMessage(). That optional routine is not used here.
7. The part number for this message part is obtained with a call to mtaDecodeMessageInfoInt().

MIME Message Decoding Simple Example Output

The following shows the output generated by the program in "Example 5-1 Decoding MIME Messages Simple Example".

```
1H: Content-type: text/plain; charset=us-ascii
1H: Content-disposition: inline
1T: This is a
1T:   test message!
2H: Content-type: application/postscript
2H: Content-transfer-encoding: base64
2H: Content-disposition: attachment; filename="a.ps"
2B: #!PS
100 100 moveto 300 300 lineto stroke
showpage
```

The Output Destination

When an optional output destination is supplied to mtaDecodeMessage(), the processed input message is subsequently written to the output destination. When conversion to MIME is requested, the output message will be the result of the conversion. Additionally, the written message will reflect any changes made by the inspection routine with **mtaDecodeMessagePartDelete()**. That routine may be used to delete an atomic part or replace the part with new, caller-supplied content.

The output destination can be either a message submission to the MTA (that is, an ongoing enqueue) or an arbitrary destination represented by a caller-supplied output routine.

Enqueue Context

When using a message enqueue context, you must do the following:

1. Supply the enqueue context along with the **MTA_DECODE_NQ** item code.
2. Specification of the message's recipient list must have already been completed with **mtaEnqueueTo()** before calling mtaDecodeMessage().
3. **mtaEnqueueFinish()** must not yet have been called for the enqueue context.

After the call to mtaDecodeMessage() has completed successfully, complete the message enqueue with **mtaEnqueueFinish()**. In the event of an error, the message submission should be cancelled with **mtaEnqueueFinish()**. mtaDecodeMessage() writes the entire message header and content. There is no need for the caller to write anything to the message's header or content.

Caller-Supplied Output Routine

To use a caller-supplied output routine (for example, **decode_write()**), supply the address of the output routine along with the **MTA_DECODE_PROC** item code to mtaDecodeMessage().

Each line passed to the output routine represents a complete line of the message to be output. The output routine must add to the line any line terminators required by the output destination (for example, carriage return, line feed pairs if transmitting over the SMTP protocol, line feed terminators if writing to a UNIX text file, and so forth).

Decode Contexts

When mtaDecodeMessage() calls either a caller-supplied inspection or output routine, it passes a decode context to those routines. Through SDK routine calls, this decode context can be queried to obtain information about the message part currently being processed, as shown in Table 5–1.

Table 5–1 Access Control Process

Message Code	Description
MTA_DECODE_ CCHARSET	The character set specified with the **CHARSET** parameter of the part's **Content-type:** header line. If the part lacks a **CHARSET** specification, then the value **us-ascii** will be returned. Obtain with mtaDecodeMessageInfoString().
MTA_DECODE_CDISP	Value of the **Content-disposition:** header line, less any optional parameters. Will be a zero length string if the part lacks a **Content-disposition:** header line. Obtain with **mtaDecodeMessageInfoString()**
MTA_DECODE_CDISP_ PARAMS	Parameter list to the **Content-disposition:** header line, if any. The parsed list is returned as a pointer to an option context. For further information, see "mtaDecodeMessageInfoParams()".
MTA_DECODE_ CSUBTYPE	The content subtype specified with the part's **Content-type:** header line (for example, **plain** for **text/plain**, **gif** for **image/gif**). Defaults to **plain** when the part lacks a **Content-type:** header line.Obtain with mtaDecodeMessageInfoString().
MTA_DECODE_CTYPE	The major content type specified with the part's **Content-type:** header line (for example, **text** for **text/plain**, **image** for **image/gif**). Defaults to **text** when the part lacks a **Content-type:** header line.Obtain with mtaDecodeMessageInfoString().
MTA_DECODE_CTYPE_ PARAMS	Parameter list to the **Content-type:** header line, if any. The parsed list is returned as a pointer to an option context. For further information, see "mtaDecodeMessageInfoParams()".
MTA_DECODE_DTYPE	Data type associated with this part. Obtain with mtaDecodeMessageInfoInt().
MTA_DECODE_PART_ NUMBER	Sequential part number for the current part. The first message part is part **0**, the second part is **1**, the third part is **2**, and so on. Obtain with mtaDecodeMessageInfoInt().

A Simple Virus Scanner Example

"Example 5-2 Decoding MIME Messages Complex Example" shows how to use the mtaDecodeMessage() routine to write an intermediate processing channel that converts messages with formats other than MIME, for example UUENCODE content, to MIME output. It then decodes the MIME message, scanning it for potentially harmful attachments. (In this example, an attachment is any message part.) Any harmful attachments are removed from the message after which it is re-enqueued for delivery. The list of harmful MIME media types and file name extensions is read from

a channel option file. An example option file for the channel is shown in "Example Option File".

In this example, the MIME **Content-type:** and **Content-disposition:** header lines are used to detect potentially harmful message attachments such as executable files. This example could be extended to also scan the content of the attachments, possibly passing the contents to a virus scanner. Further, the example could be modified to return as undeliverable any messages containing harmful attachments.

> **Note:** To configure the MTA to run this channel, see "Running Your Enqueue and Dequeue Programs". The **PMDF_CHANNEL_OPTION** environment variable must give the absolute file path to the channel's option file. Also, for a discussion on configuring special rewrite rules for re-enqueuing dequeued mail, see "Preventing Mail Loops when Re-enqueuing Mail".

For the output generated by this sample program, see "Decoding MIME Messages Complex Example Output".

After the Messaging Server product is installed, these programs can be found in the following location:

*MessagingServer_home***/examples/mtasdk/**

Some lines of code are immediately preceded by a comment of the format:

```
/* See explanatory comment N */
```

where *N* is a number. The numbers are links to some corresponding explanatory text in the section that follows this code, see "Explanatory Text for Numbered Comments in the Decoding MIME Messages Complex Example".

Example 5-2 Decoding MIME Messages Complex Example

```
/*
 *  virus_scanner_simple.c
 *
 *    Remove potentially harmful content from queued messages.
 *
 */

#include &lt;stdio.h>
#include &lt;stdlib.h>
#include &lt;string.h>
#include &lt;ctype.h>
#include "mtasdk.h"

/*
 *  A structure to store our channel options
 */
typedef struct {
     /* Produce debug output?       */
     int     debug;
     /* Unwanted MIME content types */
     char    bad_mime_types[BIGALFA_SIZE+3];
     /* Length of bmt string        */
     size_t  bmt_len;
     /* Unwanted file types         */
     char    bad_file_types[BIGALFA_SIZE+3];
     /* Length of bft string        */
     size_t  bft_len;
```

```
} our_options_t;

/*
 *  Forward declarations
 */
static void error_exit(int ires, const char *msg);
static void error_report(our_options_t *options, int ires, const
                         char *func);
static int  is_bad_mime_type(our_options_t *options, mta_decode_t
                             *dctx, char *buf, size_t maxbuflen);
static int is_bad_file_type(our_options_t *options, mta_opt_t
                            *params, const char *param_name,
                            char *buf, size_t maxbuflen);
static int  load_options(our_options_t *options);

static mta_dq_process_message_t process_message;
static mta_decode_read_t decode_read;
static mta_decode_inspect_t decode_inspect;

/*
 *  main() -- Initialize the MTA SDK, load our options, and then
 *            start the message processing loop.
 */
int main()
{
      int ires;
      our_options_t options;

      /*
      *  Initialize the MTA SDK
      *  See explanatory comment 1
      */
      if ((ires = mtaInit(0)))
          error_exit(ires, "Unable to initialize the MTA SDK");

      /*
       *  Load our channel options
       *  See explanatory comment 2
       */
      if ((ires = load_options(&amp;options)))
          error_exit(ires, "Unable to load our channel options");

      /*
       *  Now process the queued messages.  Be sure to indicate a
       *  thread stack size sufficient to accomodate message
       *  enqueue processing.
       *  See explanatory comment 3
       */
      if ((ires = mtaDequeueStart((void *)&amp;options,
                                  process_message, NULL, 0)))
          error_exit(ires, "Error during dequeue processing");

      /*
       * All done
       */
      mtaDone();
      return(0);
}

/*
```

```
 *  process_message() -- This routine is called by
 *                       mtaDequeueStart() to process each queued
 *                       message.  We dont make use of ctx2, but
 *                       ctx1 is a pointer to our channel options.
 *  See explanatory comment 4
 */
static int process_message(void **ctx2, void *ctx1, mta_dq_t *dq,
                           const char *env_from, size_t
                           env_from_len)
{
     const char *adr;
     int disp, ires;
     size_t len;
     mta_nq_t *nq;
     our_options_t *options = (our_options_t *)ctx1;

     /*
      *  Initializations
      */
     nq = NULL;

     /*
      *  A little macro to do error checking on mta*() calls
      */
#define CHECK(f,x) \
     if ((ires = x)) { error_report(options, ires, f); goto \
                       done_bad; }

     /*
      *  Start a message enqueue.  Use the dequeue context to copy
      *  envelope flags fromt the current message to this new
      *  message being enqueued.
      *  See explanatory comment 5
      */
     CHECK("mtaEnqueueStart",
           mtaEnqueueStart(&amp;nq, env_from, env_from_len,
           MTA_DQ_CONTEXT, dq, 0));

     /*
      *  Process the envelope recipient list
      *  See explanatory comment 6
      */
     while (!(ires = mtaDequeueRecipientNext(dq, &amp;adr, &amp;len, 0)))
     {
          /*
           *  Add this envelope recipient address to the message
           *  being enqueued.  Use the dequeue context to copy
           *  envelope flags for this recipient from the current
           *  message to the new message.
           */
          ires = mtaEnqueueTo(nq, adr, len, MTA_DQ_CONTEXT,
                              dq, MTA_ENV_TO, 0);
          /*  See explanatory comment 7 */
          disp = (ires) ? MTA_DISP_DEFERRED : MTA_DISP_RELAYED;
          CHECK("mtaDequeueRecipientDisposition",
                mtaDequeueRecipientDisposition(dq, adr, len,
                                               disp, 0));
     }

     /*
```

```
         *  A normal exit from the loop occurs when
         *  mtaDequeueRecipientNext() returns an MTA_EOF status.
         *  Any other status signifies an error.
         */
        if (ires != MTA_EOF)
        {
            error_report(options, ires, "mtaDequeueRecipientNext");
            goto done_bad;
         }

         /*
          *  Begin the MIME decode of the message
          *  See explanatory comment 8
          */
         CHECK("mtaDecodeMessage",
               mtaDecodeMessage(
                 /* Private context is our options */
                 (void *)options,
                 /* Input is the message being dequeued */
                 MTA_DECODE_DQ, (void *)dq,
                 /* Output is the message being enqueued */
                 MTA_DECODE_NQ, (void *)nq,
                 /* Inspection routine */
                 decode_inspect,
                 /* Convert non-MIME formats to MIME */
                 MTA_DECODE_THURMAN, 0));

         /*
          *  Finish the enqueue
          *  NOTE: ITS IMPORTANT TO DO THIS before DOING THE
          *  DEQUEUE. YOU WILL LOSE MAIL IF YOU DO THE DEQUEUE FIRST
          *  and then THE ENQUEUE FAILS.
          *  See explanatory text 9
          */
         CHECK("mtaEnqueueFinish", mtaEnqueueFinish(nq, 0));
         nq = NULL;

         /*
          *  Finish the dequeue
          */
         CHECK("mtaDequeueFinish", mtaDequeueMessageFinish(dq, 0));

         /*
          *  All done with this message
          */
         return(MTA_OK);

    done_bad:
        /*
         *  Abort any ongoing enqueue or dequeue
         */
        if (nq)
             mtaEnqueueFinish(nq, MTA_ABORT, 0);
        if (dq)
             mtaDequeueMessageFinish(dq, MTA_ABORT, 0);

        /*
         *  And return our error status
         */
        return(ires);
```

```
}

#undef CHECK

/*
 *  decode_inspect() -- This is the routine that inspects each
 *                      message part, deciding whether to accept
 *                      or reject it.
 *  See explanatory comment 10
 */
static int decode_inspect(void *ctx, mta_decode_t *dctx,
                          int data_type,const char *data,
                          size_t data_len)
{
     char buf[BIGALFA_SIZE * 2 + 10];
     int i;
     our_options_t *options = (our_options_t *)ctx;

     /*
      *  See if the part has:
      *
      *    1. A bad MIME content-type,
      *    2. A bad file name extension in the (deprecated)
      *       NAME= content-type parameter, or
      *    3. A bad file name extension in the
      *       FILENAME= content-disposition parameter.
      */
     i = 0;
     if ((i = is_bad_mime_type(ctx, dctx, buf, sizeof(buf))) ||
         is_bad_file_type(ctx,
                          mtaDecodeMessageInfoParams(dctx,
                                  MTA_DECODE_CTYPE_PARAMS, NULL),
                          "NAME", buf, sizeof(buf)) ||
         is_bad_file_type(ctx,
                          mtaDecodeMessageInfoParams(dctx,
                                  MTA_DECODE_CDISP_PARAMS, NULL),
                          "FILENAME", buf, sizeof(buf)))
     {
         char msg[BIGALFA_SIZE*4 + 10];

         /*
          *  Replace this part with a text message indicating
          *  that the parts content has been deleted.
          *  See explanatory comment 11
          */
         if (i)
              i = sprintf(msg,
         "The content of this message part has been removed.\n"
         "It contained a potentially harmful media type of %.*s",
                          strlen(buf)-2, buf+1);

         else
              i = sprintf(msg,
      "The content of this message part has been removed.\n"
      "It contained a potentially harmful file named '%s'", buf);
         return(mtaDecodeMessagePartDelete(dctx,
                               MTA_REASON, msg, i,
                               MTA_DECODE_CTYPE, "text", 4,
                               MTA_DECODE_CSUBTYPE, "plain", 5,
                               MTA_DECODE_CCHARSET, "us-ascii", 8,
```

```
                              MTA_DECODE_CDISP, "inline", 6,
                              MTA_DECODE_CLANG, "en", 2, 0));
     }
     else
          /*
           *  Keep the part
           *  See explanatory comment 12
           */
          return(mtaDecodeMessagePartCopy(dctx, 0));
}

/*
 *  is_bad_mime_type() -- See if the parts media type is in our
 *                        bad MIME content types, for example:
 *                        application/vbscript
 *  See explanatory comment 13
 */
static int is_bad_mime_type(our_options_t *options,
                            mta_decode_t *dctx, char *buf,
                            size_t maxbuflen)
{
     const char *csubtype, *ctype;
     size_t i, len1, len2;
     char *ptr;

     /*
      *  Sanity checks
      */
      if (!options || !options->bmt_len ||
          !options->bad_mime_types[0] ||
          !dctx)
           return(0);

      /*
       *  Get the MIME content type
       */
      ctype = mtaDecodeMessageInfoString(dctx, MTA_DECODE_CTYPE,
                                         NULL, &amp;len1);
      csubtype = mtaDecodeMessageInfoString(dctx,
                                            MTA_DECODE_CSUBTYPE,
                                            NULL, &amp;len2);

      /*
       *  Build the string: &lt;0x01>type/subtype&lt;0x01>&lt;0x00>
       */
      ptr = buf;
      *ptr++ = (char)0x01;
      for (i = 0; i &lt; len1; i++)
           *ptr++ = tolower(*ctype++);
      *ptr++ = /;
      for (i = 0; i &lt; len2; i++)
           *ptr++ = tolower(*csubtype++);
      *ptr++ = (char)0x01;
      *ptr = \0;

      /*
       *  Now see if the literal just built occurs in the list of
       *  bad MIME content types
       */
      return((strstr(options->bad_mime_types, buf)) ? -1 : 0);
```

```
}

/*
 *  is_bad_file_type() -- See if the part has an associated file
 *                        name whose file extension is in our list
 *                        of bad file names, such as .vbs.
 *  See explanatory comment 14
 */
static int is_bad_file_type(our_options_t *options,
                            mta_opt_t *params,
                            const char *param_name, char *buf,
                            size_t maxbuflen)
{
      const char *ptr1;
      char fext[BIGALFA_SIZE+2], *ptr2;
      size_t i, len;

      /*
       *  Sanity checks
       */
      if (!options || !options->bft_len || !params || !param_name)
           return(0);

      len = 0;
      buf[0] = \0;
      if (mtaOptionString(params, param_name, 0, buf, &amp;len,
                          maxbuflen - 1) ||
          !len || !buf[0])
          /*
           *  No file name parameter specified
           */
          return(0);

          /*
           *  A file name parameter was specified.  Parse it to
           *  extract the file extension portion, if any.
           */
          ptr1 = strrchr(buf, .);
          if (!ptr1)
               /*
                *  No file extension specified
                */
               return(0);

          /*
           *  Now store the string created earlier in fext[]
           *  Note that we drop the . from the extension.
           */
          ptr1++; /* Skip over the . */
          ptr2 = fext;
          *ptr2++ = (char)0x01;
          len = len - (ptr1 - buf);
          for (i = 0; i &lt; len; i++)
               *ptr2++ = tolower(*ptr1++);
          *ptr2++ = (char)0x01;
          *ptr2++ = \0;

          /*
           *  Now return -1 if the string occurs in
           *  options->bad_file_types.
```

```
          */
         return((strstr(options->bad_file_types, fext))
                ? -1 : 0);
}

/*
 *  load_options() -- Load our channel options from the channels
 *                    option file
 *  See explanatory comment 15
 */
static int load_options(our_options_t *options)
{
     char buf[BIGALFA_SIZE+1];
     size_t buflen, i;
     mta_opt_t *channel_opts;
     int ires;
     const char *ptr0;
     char *ptr1;

     /*
      *  Initialize the our private channel option structure
      */
     memset(options, 0, sizeof(our_options_t));

     /*
      *  Access the channels option file
      *  See explanatory comment 16
      */
     channel_opts = NULL;
     if ((ires = mtaOptionStart(&amp;channel_opts, NULL, 0, 0)))
     {
          mtaLog("Unable to access our channel option file");
          return(ires);
     }

     /*
      *  DEBUG=0|1
      */
     options->debug = 0;
     mtaOptionInt(channel_opts, "DEBUG", 0, &amp;options->debug);
     if (options->debug)
         mtaDebug(MTA_DEBUG_SDK, 0);

     /*
      *  BAD_MIME_TYPES=type1/subtype1[,type2/subtype2[,...]]
      */
     buf[0] = \0;
     mtaOptionString(channel_opts, "BAD_MIME_TYPES", 0, buf,
                     &amp;buflen, sizeof(buf));

     /*
      *  Now translate the comma separated list:
      *
      *    Type1/Subtype1[,Type2/Subtype2[,...]]
      *
      *  to
      *
      *&lt;0x01>type1/subtype1[&lt;0x01>type2/subtype2[&lt;0x01>...]]&lt;0x01>
      */
```

```
ptr0 = buf;
ptr1 = options->bad_mime_types;
*ptr1++ = (char)0x01;
for (i = 0; i &lt; buflen; i++)
{
     if (*ptr0 != ,)
         *ptr1++ = tolower(*ptr0++);
     else
     {
          *ptr1++ = (char)0x01;
          ptr0++
      }
}
*ptr1++ = (char)0x01;
*ptr1   = \0;
options->bmt_len = ptr1 - options->bad_mime_types;

/*
 *  BAD_FILE_TYPES=["."]Ext1[,["."]Ext2[,...]]
 */
buf[0] = \0;
buflen = 0;
mtaOptionString(channel_opts, "BAD_FILE_TYPES", 0, buf,
                &amp;buflen, sizeof(buf));

/*
 *  Now translate the comma separated list:
 *    ["."]Ext1[,["."]Ext2[,...]]
 *
 *  to
 *
 *    &lt;0x01>ext1[&lt;0x01>ext2[&lt;0x01>...]]&lt;0x01>
 */
ptr0 = buf;
ptr1 = options->bad_file_types;
*ptr1++ = (char)0x01;
for (i = 0; i &lt; buflen; i++)
{
     switch(*ptr0)
    {
     default :   /* copy after translating to lower case */
          *ptr1++ = tolower(*ptr0++);
          break;
     case . :  /* discard */
          break;
     case , :  /* end current type */
          *ptr1++ = (char)0x01;
          ptr0++;
          break;
     }
}
*ptr1++ = (char)0x01;
*ptr1   = \0;
options->bft_len = ptr1 - options->bad_file_types;

/*
 *  Dispose of the mta_opt_t context
 *  See explanatory comment 17
 */
mtaOptionFinish(channel_opts);
```

```
     /*
      *  And return a success
      */

     return(MTA_OK);
}

/*
 *  error_report()  Report an error condition when debugging is
 *                    enabled.
 */
static void error_report(our_options_t *options, int ires,
                         const char *func)
{
     if (options->debug)
         mtaLog("%s() returned %d; %s",
                (func ? func : "?"), ires, mtaStrError(ires));
}

/*
 *  error_exit() -- Exit with an error status and error message.
 */
static void error_exit(int ires, const char *msg)
{
     mtaLog("%s%s%s", (msg ? msg : ""), (msg ? "; " : ""),
            mtaStrError(ires));
     exit(1);
}
```

Example Option File

This example lists the MIME media types and file extensions this program is to consider potentially harmful.

```
DEBUG=1
BAD_MIME_TYPES=application/vbscript
BAD_FILE_TYPES=bat,com,dll,exe,vb,vbs
```

Sample Input Message

The example that follows is the text of a sample input message the program in "Example 5-2 Decoding MIME Messages Complex Example" is to process. The second message part is a file attachment. The attached file name is **trojan_horse.vbs**. Consequently when this message is processed by the channel, it should remove the attachment as the file extension **.vbs** is in the list of harmful file extensions. The sample program replaces the attachment with a text attachment indicating the content has been deleted.

```
Received: from [129.153.12.22] ([129.153.12.22])
 by frodo.siroe.com (Sun Java System Messaging Server 6 2004Q2 (built Apr 7
 2003)) with SMTP id &lt;0HD7001023OYDA00@frodo.siroe.com\> for
 for sue@sesta.com; Fri, 11 Apr 2003 13:03:23 -0700 (PDT)
Date: Fri, 11 Apr 2003 13:03:08 -0700
From: sue@sesta.com
Subject: test message
Message-id: &lt;0HD7001033P1DA00@frodo.siroe.com\>
Content-type: multipart/mixed; boundary=BoundaryMarke

--BoundaryMarker
Content-type: text/plain; charset=us-ascii
```

```
Content-disposition: inline

This is a
  test message!

--BoundaryMarker
Content-type: application/octet-stream
Content-disposition: attachment; filename="trojan_horse.vbs"
Content-transfer-encoding: base64

IyFQUwoxMDAgMTAwIG1vdmV0byAzMDAgMzAwIGxpbmV0byBzdHJva2UKc2hvd3Bh
Z2UK

--BoundaryMarker--
```

Explanatory Text for Numbered Comments in the Decoding MIME Messages Complex Example

1. The MTA SDK is explicitly initialized. This call is not really necessary as the MTA SDK will implicitly initialize itself when **mtaDequeueStart()** is called. However, for debugging purposes, it can be useful to make this call at the start of a program so that an initialization failure will show clearly in the diagnostic output. If the call is omitted, initialization failure will be less obvious. The failure will still be noted in the diagnostic output, but it will be obscured through the routine call that triggered implicit initialization.
2. Channel options are loaded via a call to the **load_options()** routine. That routine is part of this example and, as discussed later, uses the SDK routines for obtaining channel option values from the channel's option file.
3. The message dequeue processing loop is initiated with a call to **mtaDequeueStart()**.
4. For each queued message to be processed, **process_message()** will be called by **mtaDequeueStart()**.
5. A message enqueue is started. This enqueue is used to re-enqueue the queued message currently being processed. As the message is processed, its non-harmful content will be copied to the new message being enqueued.
6. The envelope recipient list is copied from the queued message to the new message being enqueued.
7. Since this is an intermediate channel, that is, it doesn't effect final delivery of a message, successful processing of a recipient address is associated with a disposition of **MTA_DISP_RELAYED**.
8. After processing the message's envelope, mtaDecodeMessage() is invoked to decode the message, breaking it into individual MIME message parts. **mtaDecodeMessage()** is told to use the current dequeue context as the input source for the message to decode. This supplies the queued message being processed as input to the MIME decoder. Further, the current enqueue context is supplied as the output destination for the resulting message. This directs mtaDecodeMessage() to output the resulting parsed message to the message being enqueued, less any harmful attachments that are explicitly deleted by the inspection routine. The routine **decode_inspect()** is supplied as the inspection routine. If the call to mtaDecodeMessage() fails, the **CHECK()** macro causes the queued message to be deferred and the message enqueue to be cancelled.
9. After a successful call to mtaDecodeMessage(), the message enqueue is committed. It is important that this be done before committing the dequeue. If the

operation is done in the other order- dequeue finish followed by enqueue finish- then mail may be lost. For example, the message would be lost if the dequeue succeeds and then deletes the underlying message file before the enqueue, and then the enqueue fails for some reason, such as insufficient disk space.

10. The inspection routine, **decode_inspect()**. This routine checks the MIME header lines of each message part for indication that the part may contain harmful content.
11. Message parts with harmful content are discarded with a call to **mtaDecodeMessagePartDelete()**. The discarded message part is replaced with a text message part containing a warning about the discarded harmful content.
12. Message parts with safe content are kept by copying them to the output message with **mtaDecodeMessagePartCopy()**.
13. Using the configured channel options, this routine determines if a message part's media type is in the list of harmful types.
14. Using the configured channel options, this routine determines if a filename appearing in the MIME header lines has an extension considered harmful.
15. The **load_options()** routine is used to load the channel's site-configured options from a channel option file.
16. The channel option file, if any, is opened and read by **mtaOptionStart()**. Since an explicit file path is not supplied, the file path specified with the **PMDF_CHANNEL_OPTION** environment variable gives the name of the option file to read.
17. After loading the channel's options, the option file context is disposed of with a call to **mtaOptionFinish()**.

Decoding MIME Messages Complex Example Output

The example that follows shows the output generated by the MIME decoding program found in "Example 5-2 Decoding MIME Messages Complex Example".

```
Received: from sesta.com by frodo.siroe.com
 Sun Java System Messaging Server Version 6 2004 Q2(built Apr 7 2003))
 id &lt;0HDE00C01BFK6500@frodo.siroe.com\> for sue@sesta.com; Tue, 11
 Apr 2003 13:03:29 -0700 (PDT)
Received: from [129.153.12.22] ([129.153.12.22])
 by frodo.siroe.com (Sun Java System Messaging Server 6 2004 Q2 (built Apr 7
 2003)) with SMTP id &lt;0HD7001023OYDA00@frodo.siroe.com\> for
 sue@sesta.com; Fri, 11 Apr 2003 13:03:23 -0700 (PDT)
Date: Fri, 11 Apr 2003 13:03:08 -0700
From: sue@sesta.com
Subject: test message
To: sue@sesta.com
Message-id: &lt;0HD7001033P1DA00@frodo.siroe.com\>
Content-type: multipart/mixed;
 boundary="Boundary_(ID_XIIwKLBET2/DDbPzRI7yzQ)"

--Boundary_(ID_XIIwKLBET2/DDbPzRI7yzQ)
Content-type: text/plain; charset=us-ascii
Content-disposition: inline

This is a
 test message!

--Boundary_(ID_XIIwKLBET2/DDbPzRI7yzQ)
Content-type: text/plain; charset=us-ascii
```

```
Content-language: en
Content-disposition: inline

The content of this message part has been removed.
It contained a potentially harmful file named "trojan_horse.vbs"

--Boundary_(ID_XIIwKLBET2/DDbPzRI7yzQ)--
```

6

MTA SDK Reference

The Oracle Communications Messaging Server MTA SDK consists of numerous routines used to facilitate the enqueuing and dequeuing of messages. This chapter contains definitions of all of the SDK routines.

Summary of SDK Routines

This sections contains a series of tables, one for each of the following logical groups of commands:

- Address Parsing
- Dequeue
- Enqueue
- Error Handling
- Initialization
- Logging and Diagnostics
- MIME Parsing and Decoding
- Miscellaneous
- Option File Processing

Each table lists the routines that comprise the group and gives a brief description of each.

Address Parsing

Table 6–1 shows the address parsing routines used to parse and extract message addresses.

Table 6–1 SDK Address Parsing

Routine Name	Description
mtaAddressFinish()	Dispose of an address context
mtaAddressGetN()	Extract the Nth individual address from a list of parsed addresses
mtaAddressParse()	Parse a list of addresses, producing an address context

Dequeue

Table 6–2 shows the dequeue routines used for dequeuing messages.

Table 6–2 SDK Dequeuing

Routine Name	Description
mtaDequeueInfo()	Obtain information about a queued message
mtaDequeueLineNext()	Obtain the next message line from a queued message
mtaDequeueMessageFinish()	Complete or cancel a message dequeue
mtaDequeueRecipientDisposition()	Set the disposition of a recipient address
mtaDequeueRecipientNext()	Obtain the next recipient address from a queued message
mtaDequeueRewind()	Move the read point for a queued message back to the start of its outermost header
mtaDequeueStart()	Begin processing queued messages
mtaDequeueThreadId()	Return the thread ID associated with the specified dequeue context.

Enqueue

Table 6–3 shows the enqueue routines used for enqueuing messages.

Table 6–3 SDK Enqueuing

Routine Name	Description
mtaEnqueueCopyMessage()	Copy a message from a dequeue context
mtaEnqueueFinish()	Complete or cancel a message submission
mtaEnqueueInfo()	Obtain information about a message submission
mtaEnqueueStart()	Begin a message submission
mtaEnqueueTo()	Add recipients to a message
mtaEnqueueWrite()	Output a line to the message header or body
mtaEnqueueWriteLine()	Output a line to the message header or body

Error Handling

Table 6–4 shows the error handling routines used for error status retrieval.

Table 6–4 SDK Error Handling

Routine Name	Description
mtaErrno()	Obtain the value of the last error status for this thread
mtaStrError()	Map an error status code to a printable string

Initialization

Table 6–5 shows the routines used for initialization.

Table 6–5 SDK Initialization

Routine Name	Description
mtaDone()	Release resources used by the MTA SDK

Table 6–5 (Cont.) SDK Initialization

Routine Name	Description
mtaInit()	Initialize the MTA SDK

Logging and Diagnostics

Table 6–6 shows the logging and diagnostics routines used to write diagnostic messages to debug log files.

Table 6–6 SDK Logging and Diagnostics

Routine Name	Description
mtaDebug()	Write internal diagnostic information to the debug log file
mtaLog()	Write to the debug log file
mtaLogv()	Write to the debug log file

MIME Parsing and Decoding

Table 6–7 shows the routines used to parse and decode a MIME formatted message.

Table 6–7 SDK MIME Parsing and Decoding

Routine Name	Description
mtaDecodeMessage()	Decode a MIME formatted message; can also convert non-MIME formats to MIME
mtaDecodeMessagePartCopy()	Copy a message part
mtaDecodeMessagePartDelete()	Delete a message part
mtaDecodeMessageInfoInt()	Obtain the value of an integer-valued parameter
mtaDecodeMessageInfoString()	Obtain the value of a string-valued parameter
mtaDecodeMessageInfoParams()	Obtain the **Content-type** or **Content-disposition** parameter list

Miscellaneous

Table 6–8 shows the routines used for miscellaneous tasks.

Table 6–8 SDK Miscellaneous

Routine Name	Description
mtaAccountingLogClose()	Close the MTA accounting log file
mtaAddressToChannel()	Determine which channel an address rewrites to
mtaBlockSize()	Obtain the value of the MTA **BLOCK_SIZE** option
mtaChannelGetName()	Obtain the channel name for the running program
mtaChannelToHost()	Determine the host name associated with a channel
mtaDateTime()	Generate a date-time string for use in an RFC 822 **Date:** header line
mtaPostmasterAddress()	Obtain the postmaster's address

Table 6–8 (Cont.) SDK Miscellaneous

Routine Name	Description
mtaStackSize()	Obtain the minimum thread stack size needed for arbitrary SDK operations
mtaUniqueString()	Generate a unique string
mtaVersionMajor()	Obtain the major version number of the MTA SDK
mtaVersionMinor()	Obtain the minor version number of the MTA SDK
mtaVersionRevision()	Obtain the revision number of the MTA SDK

Option File Processing

Table 6–9 lists the routines used to process option files and gives a brief description of each.

Table 6–9 SDK Option File Processing Routines

Routine Name	Description
mtaOptionStart()	Open and read a channel option file
mtaOptionInt()	Obtain the value associated with an integer-valued option
mtaOptionFloat()	Obtain the value associated with a real-valued option
mtaOptionString()	Obtain the value associated with a string-valued option
mtaOptionFinish()	Dispose of an option file context

MTA SDK Routines

This section describes each MTA SDK routine, including its syntax, arguments and return values, and gives a description of the routine. Table 6–10 lists the routines in alphabetical order, as they are found in this section:

Table 6–10 MTA SDK Routines

Routine Name and Page
mtaAccountingLogClose()
mtaAddressFinish()
mtaAddressGetN()
mtaAddressParse()
mtaAddressToChannel()
mtaBlockSize()
mtaChannelGetName()
mtaChannelToHost()
mtaDateTime()
mtaDebug()
mtaDecodeMessage()
mtaDecodeMessageInfoInt()
mtaDecodeMessageInfoParams()
mtaDecodeMessageInfoString()

Table 6–10 (Cont.) MTA SDK Routines

Routine Name and Page
mtaDecodeMessagePartCopy()
mtaDecodeMessagePartDelete()
mtaDequeueInfo()
mtaDequeueLineNext()
mtaDequeueMessageFinish()
mtaDequeueRecipientDisposition()
mtaDequeueRecipientNext()
mtaDequeueRewind()
mtaDequeueStart()
mtaDequeueThreadId()
mtaDone()
mtaEnqueueCopyMessage()
mtaEnqueueError()
mtaEnqueueFinish()
mtaEnqueueInfo()
mtaEnqueueStart()
mtaEnqueueTo()
mtaEnqueueWrite()
mtaEnqueueWriteLine()
mtaErrno()
mtaInit()
mtaLog()
mtaLogv()
mtaOptionFinish()
mtaOptionFloat()
mtaOptionInt()
mtaOptionStart()
mtaOptionString()
mtaPostmasterAddress()
mtaStackSize()
mtaStrError()
mtaUniqueString()
mtaVersionMajor()
mtaVersionMinor()
mtaVersionRevision()

mtaAccountingLogClose()

Close the MTA accounting log file, **mail.log_current**.

Syntax

```
void mtaAccountingClose(void)
```

Arguments

None

Description

Long running programs should periodically close the MTA accounting log file with this routine. Interactive programs that use the MTA SDK should use the mtaInit() item code when initializing the SDK with mtaInit().

Return Values

None

Example

None

mtaAddressFinish()

Dispose of an address context.

Syntax

```
void mtaAddressFinish(mta_adr_t *adr_ctx);
```

Arguments

Table 6–11 lists the **mtaAddressFinish()** arguments.

Table 6–11 mtaAddressFinish() Arguments

Argument	Description
adr_ctx	An address context created by a previous call to **mtaAddressParse()**.

Description

Address contexts created with **mtaAddressParse()** must be disposed of by calling **mtaAddressFinish()**. Failure to do so will result in memory leaks.

Return Values

None

Example

None

mtaAddressGetN()

Extract an address from a list of parsed addresses.

Syntax

```
int mtaAddressGetN(mta_adr_t   *adr_ctx,
                   size_t       address_index,
                   const char **address,
                   size_t      *address_len,
                   int          elements);
```

Arguments

Figure 6–12 lists the **mtaAddressGetN()** arguments.

Table 6–12 mtaAddressGetN() Arguments

Arguments	Description
adr_ctx	An address context created by a previous call to **mtaAddressParse()**.
address_index	Index of the address to retrieve. It is an index into a list of addresses. The first address has an index of 0.
address	Pointer to receive the selected address (a pointer to a buffer within the address context). The address will be NULL terminated. A NULL may be passed for this call argument if you do not wish to receive the pointer.
address_len	The length in bytes of the selected address, not including any NULL terminator. NULL may be passed for this call argument if you do not wish to receive the length.
elements	A bitmask indicating which RFC 822 mailbox elements of the address to return, such as phrase, route, local-part, or domain. Any combination of these elements may be returned.

Description

This routine retrieves the Nth address from a list of parsed addresses. The list of addresses must first be parsed with **mtaAddressParse()**.

Either the entire address or just a portion of it may be retrieved.

Elements Argument

Using the nomenclature of RFC 822, an address has the following four-element format:

```
phrase &lt;@route:local-part@domain>
```

> **Note:** The **@route:** element is referred to as a **source route** and is rarely seen.

An example address with all four elements is:

```
Judy Smith &lt;@siroe.com:judy.smith@email.siroe.com>
```

The **elements** argument is a bitmask indicating which of these elements to return. The bitmask is formed by a logical OR of the following symbolic constants defined in the **mtasdk.h** header file:

- **MTA_ADDR_PHRASE**- In the example, the phrase part is **Judy Smith**.
- **MTA_ADDR_ROUTE**- In the example, the route part is **@siroe.com**.
- **MTA_ADDR_LOCAL**- In the example, the local part is **judy.smith**.
- **MTA_ADDR_DOMAIN**- In the example, the domain part is **email.siroe.com**.

For example, to select just the local and domain parts, use the following value for the **elements** argument:

```
MTA_ADDR_LOCAL | MTA_ADDR_DOMAIN
```

When a value of zero is supplied for elements the following default bitmask is assumed:

```
MTA_ADDR_ROUTE | MTA_ADDR_LOCAL | MTA_ADDR_DOMAIN
```

Address Argument

This routine returns a pointer to the retrieved address and not the address itself. This pointer is to a buffer within the address context. Each time the routine is called with the same address context, that buffer is overwritten. Therefore, care must be taken when specifying the address argument. The following code example correctly specifies how the argument should be used when multiple calls are involved:

```
mtaAddressGetN(adr_ctx, 0, &amp;ptr, NULL, MTA_ADDR_LOCAL);
strcpy(buf, ptr);
strcat(buf, "@");
mtaAddressGetN(adr_ctx, 0, &amp;ptr, NULL, MTA_ADDR_DOMAIN);
strcat(buf, ptr);
```

Alternately, it could also be coded as shown in the following code fragment:

```
mtaAddressGetN(adr_ctx, 0, &amp;ptr, NULL,
               MTA_ADDR_LOCAL | MTA_ADDR_DOMAIN);
strcpy(buf, ptr);
```

However, since the pointer points to the same buffer for each call, and is overwritten at each call, it would be incorrect to code it as shown in the following code example:

```
mtaAddressGetN(adr_ctx, 0, &amp;local, NULL, MTA_ADDR_LOCAL);
mtaAddressGetN(adr_ctx, 0, &amp;domain, NULL, MTA_ADDR_DOMAIN);
strcpy(buf, local);
strcat(buf, "@");
strcat(buf, domain);
```

Return Values

Table 6–13 lists the **mtaAddressGetN()** return values.

Table 6–13 mtaAddressGetN() Return Values

Return Value	Description
0	Normal, successful completion
MTA_BADARGS	One of the following conditions occurred:# A NULL value for the **adr_content** argument # An invalid address context # An invalid bitmask for elements |

Table 6–13 (Cont.) mtaAddressGetN() Return Values

Return Value	Description
MTA_EOF	The value supplied for the **address_index** is equal to or greater than the number of addresses in the address list.

Example

The following is a code fragment that parses and displays the individual addresses from a list of two addresses, using **mtaAddressGetN()**:

```
ires = mtaAddressParse(&amp;adr_ctx, &amp;adr_count,
        "Judy Public &lt;judy.public@siroe.com\>, sue@siroe.com",
        0, 0);
for (i = 0; i &lt; adr_count; i++)
{
    mtaAddressGetN(adr_ctx, i, &amp;ptr, NULL,
                   MTA_ADDR_LOCAL | MTA_ADDR_DOMAIN);
    printf("Address %d: %s\n", i, ptr);
}
```

mtaAddressParse()

Parse a list of comma separated RFC 822 addresses.

Syntax

```
int mtaAddressParse(mta_adr_t **adr_ctx,
                    size_t     *address_count,
                    const char *address_list,
                    size_t      address_list_len,
                    int         item_code, ...);
```

Arguments

Table 6–14 lists the **mtaAddressParse()** arguments.

Table 6–14 mtaAddressParse() Arguments

Argument	Description
adr_ctx	The address context created for the parsed list of addresses.
address_count	The number of addresses parsed.
address_list	A character string containing the list of comma separated RFC 822 addresses to be parsed. The string must be NULL terminated if a value of zero is passed for **address_list_len**.
address_list_len	The length in bytes of the string of addresses to parse, not including any NULL terminator. If a value of zero is passed for this argument, then the length of **address_list** will automatically be determined.
item_code	An optional list of item codes. The list must be terminated with an integer argument with value **0**.

Description

This routine parses a list of one or more comma separated RFC 822 addresses. The input list can be of any arbitrary length. The result of the parse is represented by an

address context and a count of the parsed addresses. Each parsed address can then be individually extracted from the parsed list with a call to **mtaAddresGetN()**. The address context should be disposed of with a call to **mtaAddressFinish()**. When there are no valid addresses in the input line, the returned context will be NULL and the count zero.

> **Note:** There are two item codes that can be used in the **item_code** argument. A NULL value can be passed for either or both of the **adr_ctx** and **address_count** arguments. When NULL is passed for both, all that is learned by calling the routine is whether or not the address list is syntactically valid.

Table 6–15 lists the item codes for this routine, their additional required arguments, and gives a description of each.

Table 6–15 mtaAddressParse() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_DOMAIN	**const char *domainsize_t domain_len**	Specify a domain name to append to short-form addresses, such as **sue**, in order to create a fully qualified address, for example, **sue@siroe.com**.It must be followed by two additional call arguments: the domain name to use and the length in bytes of that domain name. If a value of **0** is passed for the length, then the domain name must be NULL terminated.
MTA_ITEM_LIST	**mta_item_list_t *item_list**	Specify a pointer to an item list array. The array must be terminated with a final array entry with an item code value of **0**. For further information on item lists, see Item Codes and Item Lists.

Return Values

Table 6–16 lists the **mtaAddressParse()** return values.

Table 6–16 mtaAddressParse() Return Values

Return Value	Description
0	Normal, successful completion.
MTA_BADARGS	A NULL value was supplied for the **address_list** argument or an optional item code argument.
MTA_NO	Unable to parse the address list. The likely cause is that one or more addresses in the list is syntactically invalid.
MTA_NOMEM	Insufficient virtual memory.
MTA_NOSUCHITEM	An invalid item code was supplied.
MTA_STRTRUERR	Item code string argument is too long.

Example

See the code example for mtaAddressGetN() for a sample code fragment that uses **mtaAddressParse()**.

mtaAddressToChannel()

Determine which channel an address rewrites to.

Syntax

```
const char *mtaAddressToChannel(char       *channel,
                                size_t     *channel_len,
                                size_t      channel_len_max,
                                const char *address,
                                size_t      address_len,
                                int         address_type,
                                int         item_code, ...);
```

Arguments

Table 6–17 lists the **mtaAddressToChannel()** arguments.

Table 6–17 mtaAddressToChannel() Arguments

Arguments	Description
channel	A pointer to a buffer to receive the NULL terminated channel name. To avoid possible truncation of the channel name, this buffer must be at least **CHANLENGTH+1** bytes long.
channel_len	An optional pointer to a **size_t** to receive the length in bytes of the returned channel name. This length does not include the NULL terminator that terminates the channel name.
channel_len_max	The maximum size in bytes of the buffer pointed at by the **channel** argument.
address	The address to rewrite. The length of this address, not including any NULL terminator, should not exceed **ALFA_SIZE** bytes. If a value of **0** is passed for the **address_len** argument, then this string must be NULL terminated.
address_len	The length in bytes of the address string, **address**. This length does not include any NULL terminator. If a value of **0** is passed for this argument, the address string must be NULL terminated.
address_type	Indicates what type of address is being rewritten. There are two types: envelope or header. In addition it can be either forward or reverse pointing. See the description for a list of the possible values.
item_code	Reserved for future use. Presently, a value of **0** must be supplied for this argument.

Description

Use this routine to determine which channel an address rewrites to. The address to be rewritten can be an envelope or header address, and can be forward or reverse pointing. The nature of the address is specified with the **address_type** argument. Table 6–18 lists the possible values for each combination: forward pointing envelope, reversing pointing envelope, forward pointing header, reverse pointing header.

Table 6–18 mtaAddressToChannel() Address Combinations

Types of Address	Value
Forward pointing envelope address	**0**, **MTA_BCC**, **MTA_CC**,**MTA_ENV_TO**, **MTA_TO**

Table 6–18 (Cont.) mtaAddressToChannel() Address Combinations

Types of Address	Value
Reverse pointing envelope address	**MTA_ENV_FROM**
Forward pointing header address	**MTA_HDR_BCC**, **MTA_HDR_CC**, **MTA_HDR_TO**
Reverse pointing header address	**MTA_HDR_FROM**

In most cases, a value of **MTA_ENV_TO** is appropriate. Other values will typically give the same result, unless the MTA configuration has rewrite rules that are sensitive to the distinctions between the four types of addresses.

Return Values

On successful operation, the routine returns the value of the **channel** argument. In the event of an error, the value returned is NULL and the **mta_errno** variable is set with an error status code. Table 6–19 lists the error status codes and gives a description of each.

Table 6–19 mtaAddressToChannel() Error Status Codes

Error Status Codes	Description
MTA_BADARGS	There are two reasons to get this return value:# A NULL value was supplied for the address argument. # An invalid value was supplied for the **address_type**. |
MTA_FOPEN	Unable to initialize the MTA SDK; can't read one or more configuration files. Issue the following command for further information: **imsimta test -rewrite**
MTA_NO	There are two reasons to get this return value:# Unable to rewrite the supplied address. Either the address is syntactically invalid, or it does not match any channel. # Unable to initialize the MTA_SDK. Issue the following command for further information: **imsimta test -rewrite** |
MTA_NOSUCHITEM	An invalid item code was specified.
MTA_STRTRUERR	There are two reasons to get this return value:# Supplied address string is too long; length can not exceed **ALFA_SIZE** bytes. # The supplied buffer to receive the channel name is too small. |

Example

None

mtaBlockSize()

Obtain the size in bytes of an MTA block size unit.

Syntax

```
size_t mtaBlockSize(void);
```

Arguments

None

Description

The MTA measures message sizes in units of blocks. Units of blocks are used, for instance, when logging message sizes, and for the **MTA_FRAGMENT_BLOCKS** item code in the **mtaEnqueueStart()** routine. By default, a block is 1024 bytes. However, sites can change this setting with the **BLOCK_SIZE** option in the **option.dat** file.

Programs using the SDK can translate units of bytes to blocks by dividing the number of bytes by the value returned by **mtaBlockSize()**, that is:

```
bytes_per_block = mtaBlockSize();
block_limit = byte_limit / bytes_per_block;
```

Return Values

In the event of a failure, the routine returns the value zero and sets **mta_errno** with an error status code. This routine only fails when initialization of the MTA SDK fails. Table 6–20 lists the error status codes set in **mta_errno** when there is an error returned by **mtaBlockSize()**.

Table 6–20 mtaBlockSize() Error Status Codes

Error Status Codes	Description
MTA_FOPEN	Unable to initialize the MTA SDK. Unable to read one or more configuration files. Issue the following command for further information: **imsimta test -rewrite**
MTA_NO	Unable to initialize the MTA SDK. Issue the following command for further information: **imsimta test -rewrite**

Example

The following code fragment displays the MTA block size setting:

```
printf ("BLOCK_SIZE = %u\n", mtaBlockSize());
```

mtaChannelGetName()

Determine the channel name for the currently running program.

Syntax

```
const char *mtaChannelGetName(char   *channel,
                              size_t *channel_len,
                              size_t  channel_len_max);
```

Arguments

Table 6–21 lists the **mtaChannelGetName()** arguments.

Table 6–21 mtaChannelGetName() Arguments

Arguments	Description
channel	A pointer to a buffer to receive the NULL terminated channel name. To avoid possible truncation of the channel name, this buffer must be at least **CHANLENGTH+1** bytes long.
channel_len	An optional pointer to a **size_t** to receive the length in bytes of the returned channel name. This length does not include the NULL terminator that terminates the channel name.

Table 6–21 (Cont.) mtaChannelGetName() Arguments

Arguments	Description
channel_len_max	The maximum size in bytes of the buffer pointed at by the **channel** argument.

Description

A running program can discover its channel name with this routine. The channel name is typically set using the **PMDF_CHANNEL** environment variable.

Return Values

In the event of an error, the routine returns NULL. The error status code is set in **mta_errno**. Table 6–22 lists the **mtaChannelGetName()** error status codes.

Table 6–22 mtaChannelGetName() Error Status Codes

Error Status Codes	Description
MTA_BADARGS	A NULL value passed for the **channel** argument.
MTA_NO	Unable to determine the channel name from the runtime environment.
MTA_STRTRUERR	Channel buffer too small to receive the channel name. The buffer must be at least **CHANLENGTH+1** bytes long.

Example

The following code fragment uses this routine to print the channel name.

```
char buf[CHANLENGTH+1];

printf("Channel name: %s\n",
       mtaChannelGetName(buf, NULL, sizeof(buf)));
```

mtaChannelToHost()

Determine the host name associated with a channel.

Syntax

```
const char *mtaChannelToHost(char  **host,
                             size_t *host_len,
                             int     item_code, ...);
```

Arguments

Table 6–23 lists the mtaChannelToHost() arguments.

Table 6–23 mtaChannelToHost() Arguments

Arguments	Description
host	A pointer to receive the host name. The host name will be NULL terminated. NULL can be passed for this call argument.

Table 6–23 (Cont.) mtaChannelToHost() Arguments

Arguments	Description
host_len	An optional pointer to a **size_t** to receive the length in bytes of the returned host name. This length does not include the NULL terminator that terminates the host name. A value of NULL can be passed for this call argument.
item_code	An optional list of item codes. The list must be terminated with an integer argument with value **0**.

Description

This routine is used to determine the host name associated with a particular channel.

The channel name can be specified in one of three ways:

- Implicitly specified. For this case, no item codes other than the terminating **0** are specified and the channel name is the one for the running program. The channel name is set using the **PMDF_CHANNEL** environment variable.
- Explicitly specified with the **MTA_CHANNEL** item code.
- Set using the **MTA_DQ_CONTEXT** item code, which is taken to be the channel name associated with a specified dequeue context.

In all cases, the official host name of the selected channel is the host name that is returned. The official host name for a channel is the one that appears on the second line of the channel definition in the MTA configuration file, **imta.cnf**.

Table 6–24 lists the item codes and any associated arguments.

Table 6–24 mtaChannelToHost() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ CHANNEL	**const char *channelsize_t channel_len**	Explicitly specify a channel name for the official host name. This item code must be followed by the two additional call arguments, specifying:# The channel name. # The length in bytes of that channel name. If a value of **0** is passed for the length, the channel name must be NULL terminated. |
MTA_DQ_ CONTEXT	**mta_dq_t *dq_ctx**	Use the channel associated with the specified dequeue context. This item code must be followed by one additional call argument: a pointer to a dequeue context generated by **mtaDequeueStart()**.
MTA_ITEM_ LIST	**mta_item_list_t *item_list**	Specify a pointer to an item list array that is terminated with a final array entry that has an item code value of **0**. For further information on item lists, see "Item Codes and Item Lists".

When none of the above item codes are specified, the channel name is taken from the runtime environment, using **PMDF_CHANNEL** environment variable.

On successful completion, the host name is stored in the buffer pointed at by the **host** argument, and the value of the **host** argument is returned.

Return Values

In the event of an error, **mtaChannelToHost()** will return NULL, but will set **mta_ errno**. Table 6–25 lists the error status codes for this routine.

Table 6–25 mtaChannelToHost() Error Status Codes

Error Status Codes	Description
MTA_BADARGS	A NULL value was supplied for either of these two argument:# The **host** argument in the routine call. # An argument to an item code. |
MTA_FOPEN	Unable to initialize the MTA SDK. Unable to read one or more configuration files. Issue the following command for further information: **imsimta test -rewrite**
MTA_NO	One of the following errors occurred:# Unable to determine the channel name from the runtime environment. # Unable to initialize the MTA SDK. For further information, issue the following command: **imsimta test -rewrite** |
MTA_NOSUCHCHAN	The selected channel name does not appear in the MTA configuration file, **imta.cnf**.
MTA_NOSUCHITEM	An invalid item code was specified.

Example

```
printf("Host name: %s\n",
       mtaChannelToHost(NULL, NULL, MTA_CHANNEL,
                        "tcp_local", 0, 0));
```

mtaDateTime()

Obtain the current date and time in an RFC 822 and RFC 1123 complaint format.

Syntax

```
const char *mtaDateTime(char   *date,
                        size_t *date_len,
                        size_t  date_len_max,
                        time_t  time);
```

Arguments

Table 6–26 lists the **mtaDateTime()** arguments.

Table 6–26 mtaDateTime() Arguments

Arguments	Description
date	A pointer to a buffer to receive the NULL terminated date and time string. To avoid possible truncation of the string, this buffer should be at least 81 bytes long.
date_len	An optional pointer to a **size_t** to receive the length in bytes of the returned date and time string. This length does not include the NULL terminator that terminates the host name. A value of NULL can be passed for this call argument.
date_len_max	The maximum size in bytes of the buffer pointed at by the **date** argument.
time	The date and time for which to generate the string representation. To use the current local time, pass a value of zero for this argument.

Description

This routine generates an RFC 2822 compliant date and time string suitable for use in an RFC 822 **Date:** header line. To generate a date and time string for a specific time, supply the time as the **time** argument. Otherwise, supply a value of **0** for the **time** argument and a date and time string will be generated for the current local time.

On successful completion, the date and time string is stored in the buffer pointed at by the **date** argument, and the value of the **date** argument is returned.

Return Values

In the event of an error, **mtaDateTime()** will return NULL. It will set the error status code in **mta_errno**. Table 6–27 lists the **mtaDateTime()** error status codes.

Table 6–27 mtaDateTime() Error Status Codes

Error Status Codes	Description
MTA_BADARGS	A value of NULL was supplied for the **date** argument.
MTA_STRTRU	The **date** buffer is too small; the returned value has been truncated to fit.

Example

```
char buf[80+1];

printf("The current date and time is %s\n",
       mtaDateTime(buf, NULL, sizeof(buf), (time_t)0);
```

mtaDebug()

Enable generation of MTA SDK diagnostic output.

Syntax

```
int mtaDebug(int item_code, ...);
```

Arguments

Table 6–28 lists the **mtaDebug()** arguments.

Table 6–28 mtaDebug() Arguments

Arguments	Description
item_code	An optional list of item codes. The list must be terminated with an integer argument with value **0**.

Description

Many of the low-level MTA subroutine libraries can produce diagnostic output as can the MTA SDK itself. This output, when enabled, is directed to **stdout**. When a channel program is run by the Job Controller, **stdout** is directed to the channel's debug log file. Use this diagnostic output when developing programs.

Note: **mtaDebug()** may also be used in production programs; however, caution should be used, as it can be quite verbose and voluminous, thereby degrading performance and consuming disk space.

As described in Table 6–29, item codes are used to select specific types of diagnostic output.

Table 6–29 mtaDebug() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ DEBUG_ DECODE	None	Enable diagnostic output from the low-level MIME decoding routines. This might be helpful when trying to understand MIME conversions that occur either when enqueuing messages (and the destination channel is configured to invoke MIME conversions, for example, marked with channel options such a **thurman** or **inner**, or when using **mtaDecodeMessage.()**
MTA_ DEBUG_ DEQUEUE	None	Enable diagnostic output from low-level queue processing routines. Use this when trying to understand issues around reading and processing of queued message files. This will not help diagnose the selection of queued messages, which is handled by the Job Controller.Enabling this diagnostic output is the equivalent of setting **DEQUEUE_DEBUG=1** in the option file, **option.dat**.
MTA_ DEBUG_ ENQUEUE	None	Enables output from low-level message enqueue routines. Can be used to diagnose the address rewriting process, destination channel selection, header processing, and other types of processing that occurs when a message is enqueued.Enabling this diagnostic output is the equivalent of setting **MM_DEBUG=5** in the **option.dat** file.
MTA_ DEBUG_MM	**size_t level**	Enable diagnostic output from the low-level message enqueue routines. The item code must be followed by one additional call argument: the debug level to use. The value of level ranges from **0** to **20**. Enqueue diagnostics can be used to diagnose the address rewriting process, destination channel selection, header processing and other types of processing that occurs when a message is enqueued.Enabling this diagnostic output is equivalent to setting **DEQUEUE_DEBUG=level** in the **option.dat** file.
MTA_ DEBUG_OS	None	Enable diagnostic output from the low-level operating system dependent routines. This output is helpful when diagnosing problems associated with creating, opening, writing, or reading files. This typically happens when attempting to enqueue messages, which requires permissions to create and write messages in the MTA queues.Enabling this output is equivalent to setting **OS_ DEBUG=1** in the **option.dat** file.
MTA_ DEBUG_SDK	None	Enable diagnostic output for the MTA SDK. When this is enabled, diagnostic information will be output whenever the SDK returns an error result.
MTA_ITEM_ LIST	mta_item_list_ t *item_list	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of **0**. For further information on item lists, see "Item Codes and Item Lists".

Return Values

Table 6–30 mtaDebug() Return Values

Return Values	Description
0	Successful, normal completion.
MTA_BADARGS	A NULL value was supplied for a pointer to an item list array.
MTA_FOPEN	Unable to initialize the MTA SDK. Unable to read one or more configuration files. For further information, issue the following command: **imsimta test -rewrite**
MTA_NO	Unable to initialize the MTA SDK. For further information issue the following command: **imsimta test -rewrite**
MTA_NOSUCHITEM	An invalid item code was specified.

Example

```
mtaDebug(MTA_DEBUG_SDK, MTA_MM_DEBUG, 8, 0);
```

mtaDecodeMessage()

Decode a MIME formatted message; optionally convert non-MIME formats to MIME.

Syntax

```
int mtaDecodeMessage(void                *ctx,
                     int                  input_type,
                     void                *input,
                     int                  output_type,
                     void                *output,
                     mta_decode_inspect_t *inspect,
                     int                  item_code, ...);
```

Arguments

Table 6–31 lists the **mtaDecodeMessage()** arguments.

Table 6–31 mtaDecodeMessage() Arguments

Arguments	Description
ctx	Optional pointer to a caller-supplied context or other data type. This pointer will be passed as the **ctx** argument to any caller-supplied routines, such as the one supplied as the **inspect** argument. A value of NULL can be passed for this argument.
input_type	Input type indicator describing the input source to use, either a dequeue context or a caller-supplied routine. There are only two valid values: **MTA_DECODE_DQ**, **MTA_DECODE_PROC**.
input	For **input_type=MTA_DECODE_DQ**, input must be a pointer to a dequeue context created by **mtaDequeueStart()**.For **input_type=MTA_DECODE_PROC**, input must be the address of a routine of type **mta_decode_read_t**.
output_type	Optional output type indicator describing the output destination to use, either an enqueue context or a caller-supplied routine. Valid values are: **0**, **MTA_DECODE_NQ**, **MTA_DECODE_PROC**. When a value of **0** is supplied, the **output** argument is ignored.

Table 6–31 (Cont.) mtaDecodeMessage() Arguments

Arguments	Description
output	For **output_type=MTA_DECODE_NQ**, output must be a pointer to an enqueue context created with **mtaEnqueueStart()**.For **output_type=MTA_DECODE_PROC**, output must be the address of a routine to type **mta_decode_write_t**. This argument is ignored when a value of **0** is supplied for **output_type**.
inspect	The address of an inspection routine of type **mta_decode_inspect_t**.
item_code	An optional list of item codes. The list must be terminated with an integer argument with value **0**.

Description

The MTA has powerful facilities for parsing and decoding single and multipart messages formatted using the MIME Internet messaging format. Additionally, these facilities can convert messages with other formats to MIME, for example, text parts with BINHEX or UUENCODE data, the RFC 1154 format, and many other proprietary formats. The **mtaDecodeMessage()** routine provides access to these facilities, parsing either a queued message or a message from an arbitrary source such as a disk file or a data stream.

There are two usage modes for **mtaDecodeMessage()**. In the first mode, messages are simply parsed, any encoded content decoded, and each resulting, atomic message part presented to an inspection routine. This mode of usage is primarily of use to channels that interface the MTA to non-Internet mail systems such as SMS and X.400. The second mode of operation allows the message to be rewritten after inspection by an output routine. The output destination for this rewriting may be either the MTA channel queues, or an arbitrary destination via a caller-supplied output routine.

During the inspection process in this second usage mode, individual, atomic message parts may be discarded or replaced with text. This operational mode is primarily of use to intermediate processing channels that need to scan message content or perform content conversions, for example, virus scanners and encryption software.

"Example 5-1 Decoding MIME Messages Simple Example" illustrates the first usage mode, while "Example 5-2 Decoding MIME Messages Complex Example" illustrates the second.

Inspection Routine

Key to either usage mode for **mtaDecodeMessage()** is the inspection routine, pointed to with the **inspect** argument. The **mtaDecodeMessage()** routine presents each atomic message part to the inspection routine one line at a time. The presentation begins with the part's header lines. Once all of the header lines have been presented, the lines of content are presented next. The following points should also be noted:

- Message parts need not have any content. A common example is a single part message with no content for which the sender used the **Subject:** header line to express their message.
- In the case of a non-multipart message, the message has a single part. The header for this sole part is the header for the message itself. As noted previously, there may or may not be any content to this single part.
- In the case of a multipart message, individual parts need not have a part header. In such cases, MIME defaults apply and imply that the content is **text/plain** using the US-ASCII character set.

- Regardless of the value of the **Content-transfer-encoding:** header line, the content presented will no longer be encoded.
- In the case of a multipart message, the outermost header is not presented. However, it may be inspected by means of an output routine. For a discussion of the output routine, see "Output Routine" that follows.

The following code fragment shows the required syntax of an inspection routine:

```
int inspection_routine(void         *ctx,
                       mta_decode_t *dctx,
                       int           data_type,
                       const char   *data,
                       size_t        data_len);
```

Table 6–32 lists each of the inspection routine's arguments, and gives a description of each.

Table 6–32 mtaDecodeMessage Routine Arguments

Arguments	Description
ctx	The caller-supplied private context.
dctx	A decode context created by **mtaDecodeMessage()**. This decode context represents the current part being processed. This context is to be used with calls to the other decode routines requiring a decode context. This context is automatically disposed of by **mtaDecodeMessage()**.
data_type	The nature of the data being presented:* For a header line: **MTA_DATA_HEADER** * For a line of text-based content: **MTA_DATA_TEXT** * For a line of binary content: **MTA_DATA_BINARY** * No data at all: **MTA_DATA_NONE**. Atomic part content with a MIME content type of **text/ or** message/ is considered to be text-based. Such content is given the data type **MTA_DATA_TEXT**. All other atomic part content (**audio/,** image/, and **application/*)** is considered to be binary and denoted by the data type **MTA_DATA_BINARY**. The data type **MTA_DATA_NONE** is only presented when using an optional output routine (supplied with the **output** argument in **mtaDecodeMessage()**). |
data	A pointer to the data being presented. Message lines will not have carriage-return or line-feed terminators, nor is the data itself NULL terminated. (However, in the case of binary data, there may be carriage returns, line feeds, or even NULLs embedded within the data itself.)
data_len	The length in bytes of the data being presented. This length may be **0**, which indicates a blank line and not the absence of any data (**MTA_DATA_NONE**).

Output Routine

When an output routine is not used, the inspection routine can detect the transition from one message part to another by observing the part number on each call. The part number is obtained by calling mtaDecodeMessageInfoString() with an item value of **MTA_DECODE_PART_NUMBER**.

When the optional output routine (pointed to by the **output** argument) is used, an additional data type, **MTA_DATA_NONE**, is presented to the inspection routine. It is presented to the inspection routine after the part's header and entire content have been presented. However, no data is actually presented for the **MTA_DATA_NONE** type. As such, this data type merely serves to (1) let the inspection routine know that the entire part has now been presented, and (2) allows the inspection routine a final

chance to delete the part from the data being output to the output routine. For example, it allows a virus scanner to be activated and a judgment passed. Based upon the result of the virus scan, the part can then either be copied to the output or not.

If it is not copying the part to the output, the inspection routine must call **mtaDecodeMessagePartDelete()**. That call will either delete the part entirely, or optionally replace it with caller-supplied content. Calling **mtaDecodeMessagePartCopy()** makes the copy operation explicit; if neither routine is called, then the part will be implicitly copied to the message being output.

When using an output routine, the inspection routine may call **mtaDecodeMessagePartDelete()** or **mtaDecodeMessagePartCopy()** at any time. It is not necessary to wait until the inspection routine is called with a data type of **MTA_DATA_NONE**. For instance, a virus scanner may want to discard a part when it sees that the part's content type indicates an executable program. However, once either of these routines is called, the inspection routine will not be called any further for that message part.

Dequeue Context

The message to be decoded is supplied by either a dequeue context or a caller-supplied input routine. When using a dequeue context, observe the following points:

- Specify **MTA_DECODE_DQ** for the **input_type** call argument.
- Pass the dequeue context from **mtaDequeueStart()** as the input argument.
- The recipient list of the message being dequeued must have already been read by **mtaDequeueRecipientNext()** before calling **mtaDecodeMessage()**.
- **mtaDequeueMessageFinish()** must not yet have been called for the dequeue context.
- After using a dequeue context with **mtaDecodeMessage()**, no further calls to **mtaDequeueRecipientNext()** can be made.
- Calls to **mtaDequeueLineNext()** can only be performed after a call to **mtaDequeueRewind()**.

Caller-Supplied Input Routine

When using a caller-supplied input routine to supply the message to be decoded, specify **MTA_DECODE_PROC** for the **input_type** argument, and pass the address of the input routine as the **input** argument.

The following code fragment shows the syntax of a caller-supplied input routine:

```
int input_routine(void       * ctx,
                  const char **line,
                  size_t     * line_len);
```

Table 6–33 lists the arguments for a caller-supplied input routine, and gives a description of each.

Table 6–33 mtaDecodeMessage() Caller-Supplied Input Routine Arguments

Arguments	Description
ctx	The caller-supplied private context.
line	A pointer to the start of the next line or section of data to return. The line or data does not need to be NULL terminated.

Table 6–33 (Cont.) mtaDecodeMessage() Caller-Supplied Input Routine Arguments

Arguments	Description
line_len	The length in bytes of the line or block of data being returned. A zero length signifies zero bytes of data. That is, a zero length does not cause **mtaMessageDecode()** to automatically determine the length by searching for a NULL terminator.

On each successful call, the input routine should return a status code of **0** (**MTA_OK**). When there is no more message data to provide, then the input routine should return **MTA_EOF**. The call that returns the last byte of data should return **0**; it is the subsequent call that must return **MTA_EOF**. In the event of an error, the input routine should return a non-zero status code other than **MTA_EOF**, such as **MTA_NO**. This will terminate the message parsing process and **mtaDecodeMessage()** will return an error.

Note: By default, each block of data must be a single line of the message. This corresponds to the **MTA_TERM_NONE** item code. If the **MTA_TERM_CR**, **MTA_TERM_CRLF**, **MTA_TERM_LF**, or **MTA_TERM_LFCR** item code is specified, then the block of data need not correspond to a single, complete line of message data It may be a portion of a line, multiple lines, or even the entire message. See "Item Codes" for information about **mtaDecodeMessage()** item codes.

Enqueue Context

The parsed message may be output either as a message enqueue or written to an arbitrary destination via a caller-supplied output routine. When using a message enqueue context, observe the following points:

- Specify **MTA_DECODE_NQ** for the **output_type** call argument.
- Pass the enqueue context from **mtaEnqueueStart()** as the output.
- Specification of the message's recipient list must have already been completed with **mtaEnqueueTo()** before calling **mtaDecodeMessage()**.
- **mtaEnqueueFinish()** must not yet have been called for the enqueue context.
- After the call to **mtaDecodeMessage()** has completed successfully, complete the message enqueue with **mtaEnqueueFinish()**.
- In the event of an error, the message submission should be cancelled, with **mtaEnqueueFinish()**.
- **mtaDecodeMessage()** will write the entire message header and content. There is no need for the caller to write anything to the message's header or content.

Caller-Supplied Output Routine

To use a caller-supplied output routine, specify the **MTA_DECODE_PROC** for the **output_type** call argument, and pass the address of the output routine as the **output** argument.

This code fragment shows the syntax of a caller-supplied output routine.

```
int output_routine(void         *ctx,
                   mta_decode_t *dctx,
                   const char  **line,
                   size_t       *line_len);
```

Table 6–34 lists the arguments for a caller-supplied output routine, and gives a description of each.

Table 6–34 mtaDecodeMessage() Caller-Supplied Output Routine Arguments

Arguments	Description
ctx	The caller-supplied private context passed as **ctx** to **mtaDecodeMessage()**.
dctx	A decode context created by **mtaDecodeMessage()**. This decode context should be used with calls to the other decode routines requiring a decode context. This context is automatically disposed of by **mtaDecodeMessage()**.
line	Pointer to a line of the message to output. This line is not NULL terminated. The line will also lack any carriage return or line feed record terminators.
line_len	The length in bytes of the message line to output. A length of **0** indicates a blank line. The maximum line length presented will be **BIGALFA_SIZE** bytes (1024 bytes).

Each line passed to the output routine represents a complete line of the message to be output. The output routine must add to the line any line terminators required by the output destination (for example, carriage return, line feed pairs if transmitting over the SMTP protocol, or line feed terminators if writing to a UNIX text file). Supplying a value of zero for the **output_type** call argument, causes the output argument to be ignored. In this case no output routine will be used.

Decode Context Queries

When **mtaDecodeMessage()** calls either a caller-supplied inspection or output routine, it passes to those routines a decode context. Through various SDK routine calls, this decode context may be queried to obtain information about the message part currently being processed.

Table 6–35 lists the informational message codes that can be obtained about a message part being processed, and gives a description of each, including the SDK routine used to obtain it.

Table 6–35 mtaDecodeMessage() Message Codes

Message Code	Description
MTA_DECODE_ CCHARSET	The character set specified with the **CHARSET** parameter of the part's **Content-type:** header line. If the part lacks a **CHARSET** specification, then the value **us-ascii** will be returned. Obtain with mtaDecodeMessageInfoString().
MTA_DECODE_CDISP	Value of the **Content-disposition:** header line, less any optional parameters. Will be a zero length string if the part lacks a **Content-disposition:** header line. Obtain with mtaDecodeMessageInfoString().
MTA_DECODE_CDISP_ PARAMS	Parameter list to the **Content-disposition:** header line, if any. The parsed list is returned as a pointer to an option context. For further information, see mtaDecodeMessageInfoParams().
MTA_DECODE_ CSUBTYPE	The content subtype specified with the part's **Content-type:** header line (for example, **plain** for **text/plain**, **gif** for **image/gif**). Defaults to **plain** when the part lacks a **Content-type:** header line.Obtain with mtaDecodeMessageInfoString().

Table 6–35 (Cont.) mtaDecodeMessage() Message Codes

Message Code	Description
MTA_DECODE_CTYPE	The major content type specified with the part's **Content-type:** header line (for example, **text** for **text/plain**, **image** for **image/gif**). Defaults to **text** when the part lacks a **Content-type:** header line.Obtain with mtaDecodeMessageInfoString().
MTA_DECODE_CTYPE_PARAMS	Parameter list to the **Content-type:** header line, if any. The parsed list is returned as a pointer to an option context. For further information, see mtaDecodeMessageInfoParams().
MTA_DECODE_DTYPE	Data type associated with this part. Obtain with mtaDecodeMessageInfoInt().
MTA_DECODE_PART_NUMBER	Sequential part number for the current part. The first message part is part **0**, the second part is **1**, the third part is **2**, and so on. Obtain with mtaDecodeMessageInfoInt().

Item Codes

Table 6–36 lists the item codes for the **item_code** argument passed to **mtaDecodeMessage()**. The list of item codes must be terminated with an item code with a value of **0**.

Table 6–36 mtaDecodeMessage() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_DECODE_LEVELS_MAX	**max_levels**	Place an upper limit on the depth of nested MIME multiparts that will be parsed. When this limit is reached no further parsing of deeper, nested multiparts is performed and the parts handed over for inspection include as text content these deeper, nested multiparts. By default, no limit is imposed. When dealing with looping notification messages, it is possible for the looping message to become deeply nested.This item code must be followed by one additional call argument whose value is the integer-valued upper limit to impose: **max_levels**.
MTA_DECODE_PARTS_MAX	**max_parts**	Place an upper limit on the total number of message parts that will be parsed. When this limit is reached, no further parsing of parts is performed. By default, no limit is imposed.This item code must be followed by one additional call argument whose value is the integer-valued part limit to impose: **max_parts**.
MTA_DECODE_THRURMAN	None	When specified, the MIME parser will first translate non-MIME formatted data to MIME. By default this translation is not performed.
MTA_ITEM_LIST	**mta_item_list_t *item_list**	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of 0. For further information on item lists, see "Item Codes and Item Lists".
MTA_TERM_CR	None	Data supplied by the input routine, pointed to by the **input** argument, uses a single byte carriage return terminator to terminate each line of message data. This option is ignored when **input_type** has the value **MTA_DECODE_DQ**.
MTA_TERM_CRLF	None	Data supplied by the input routine, pointed to by the **input** argument, uses a two byte carriage-return line-feed terminator to terminate each line of message data. This option is ignored when **input_type** has the value **MTA_DECODE_DQ**.

Table 6–36 (Cont.) mtaDecodeMessage() Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_TERM_LF	None	Data supplied by the input routine, pointed to by the **input** argument, uses a single byte line-feed terminator to terminate each line of message data. This option is ignored when **input_type** has the value **MTA_DECODE_DQ**.
MTA_TERM_LFCR	None	Data supplied by the input routine, pointed to by the **input** argument, uses a two byte line-feed carriage-return terminator to terminate each line of message data. This option is ignored when **input_type** has the value **MTA_DECODE_DQ**.
MTA_TERM_NONE	None	Data supplied by the input routine, pointed to by the **input** argument, uses no line terminators. Each call to the input routine returns a single, complete line of message data. This option is ignored when **input_type** has the value **MTA_DECODE_DQ**.

Return Values

Table 6–37 mtaDecodeMessage() Return Values

Return Values	Description
0	Successful, normal completion.
MTA_BADARGS	Two conditions cause this error:# A NULL value was supplied for input, output (when output_type is non-zero), or a required argument to an item code. # An invalid value supplied for either **input_type** or **output_type**. |
MTA_FOPEN	Unable to initialize the MTA SDK. Unable to read one or more configuration files. For further information issue the following command: **imsimta text -rewrite**
MTA_NO	Can be sent for one of three reasons:# Error parsing the supplied message. # An error reading from the queued message file when **MTA_DECODE_DQ** is supplied for **input_type**. # Unable to initialize the MTA SDK. In this case, issue the command: **imsimta test -rewrite** |
MTA_NOMEM	Insufficient virtual memory.
MTA_NOSUCHITEM	An invalid item code was specified.

Example

For examples of using **mtaDecodeMessage**, see "Example 5-1 Decoding MIME Messages Simple Example" and "Example 5-2 Decoding MIME Messages Complex Example".

mtaDecodeMessageInfoInt()

Obtain integer-valued information relating to the current message part.

Syntax

```
int mtaDecodeMessageInfoInt(mta_decode_t *dctx,
                            int           item);
```

Arguments

Table 6–38 lists the **mtaDecodeMessageInfoInt()** arguments.

Table 6–38 mtaDecodeMessageInfoInt() Arguments

Arguments	Description
dctx	A decode context created by **mtaMessageDecode()**.
item	Item identifier specifying which value to return. See the description that follows for the list of permitted values for this argument.

Description

This routine is used to obtain integer-valued information about the current message part. (When **mtaDecodeMessage()** calls either a user-supplied inspection or output routine, it provides a decode context describing the current message part being processed.)

Table 6–39 lists the values for the **item** argument, and gives a description of each.

Table 6–39 mtaDecodeMessageInfoInt() Item Argument Values

Values	Description
MTA_DECODE_DTYPE	Data type associated with this part. The returned values will be **MTA_DATA_NONE**, **MTA_DATA_HEADER**, **MTA_DATA_TEXT**, or **MTA_DATA_BINARY**.
MTA_DECODE_PART_NUMBER	Sequential part number for the current part. The first message part is part **0**, the second part is **1**, the third part is **2**, and so on.

Return Values

Upon normal, successful completion the value of the requested item is returned. In the event of an error, a value of **-1** is returned and **mta_errno** is set to indicate the error status code. Table 6–40 lists the error status codes for this routine, and gives an example of each.

Table 6–40 mtaDecodeMessageInfoInt() Error Status Codes

Error Status Codes	Description
MTA_BADARGS	A NULL value was supplied for the **dctx** call argument.
MTA_NOSUCHITEM	An invalid value was supplied for the **item** call argument.

Example

```
part_number = mtaDecodeMessageInfoInt(dctx, MTA_PART_NUMBER);
```

mtaDecodeMessageInfoParams()

Obtain an option context describing the current message part's content parameters.

Syntax

```
mta_opt_t *mtaDecodeMessageInfoParams(mta_decode_t *dctx,
                                      int           item,
                                      mta_opt_t   **params);
```

Arguments

Table 6–41 lists the mtaDecodeMessageInfoParams() arguments.

Table 6–41 mtaDecodeMessageInfoParams() Arguments

Arguments	Description
dctx	A decode context created by **mtaMessageDecode()**.
item	Item identifier specifying which content parameter list to return. See the description that follows for the list of permitted values for this argument.
params	An optional pointer to receive the address of the option context describing the requested parameter list.

Description

This routine returns the parameter lists for either the **Content-type:** or **Content-disposition:** header lines. (When **mtaDecodeMessage()** calls either a user-supplied inspection or output routine, it provides a decode context describing the current part being processed.)

Table 6–42 lists the values for the item argument, and gives a description of each.

Table 6–42 mtaDecodeMessageInfoParam(s) Item Argument Values

Values	Description
MTA_DECODE_CDISP_PARAMS	Parameters associated with the **Content-disposition:** header line, if any.
MTA_DECODE_CTYPE_PARAMS	Parameters associated with the **Content-type:** header line, if any.

The option context returned upon normal completion does not need to be disposed of with **mtaOptionFinish()**. It will automatically be disposed of by **mtaDecodeMessage()**. The values of individual parameters can be queried using **mtaOptionString()**, **mtaOptionInt()**, and **mtaOptionFloat()**. Program code need not worry about whether the underlying header line exists in the parts header. If it does not, then calls to obtain individual parameter values will succeed, but return no value.

Note: If the **Content-type:** header line is not present, **mtaOptionString()** returns an empty string. This is in contrast to what happens when **mtaDecodeMessageInfoString()** is used. It always returns a value for the **CHARSET** parameter of the **Content-type:** header line. If the **Content-type:** header line is not present, it returns the MIME default value **us-ascii**.

It is important to note that the option contexts returned by this routine are only valid during the lifetime of the associated decode context. They are not valid after inspection or output of a new message part begins, nor are they valid after **mtaDecodeMessage()** returns.

Return Values

Upon normal, successful completion, a pointer to an option context is returned. In the event of an error, a NULL value is returned, and **mta_errno** is set to indicate the error status code. Table 6–43 lists the error status codes, and gives a description of each.

Table 6–43 mtaDecodeMessageInfoParams() Error Status Codes

Error Status Codes	Description
MTA_BADARGS	A NULL value was supplied for the **dctx** call argument, or an invalid decode context was supplied for **dctx**.
MTA_NOSUCHITEM	An invalid value was supplied for the **item** call argument.

Example

```
char buf[64];

strcpy(buf, "us-ascii");
mtaOptionString(
    mtaDecodeMessageInfoParams(dctx, MTA_DECODE_CTYPE_PARAMS,
    NULL), "charset", 0, buf, NULL, sizeof(buf));
printf("Message part&rsquo;s character set is %s\n", buf);
```

mtaDecodeMessageInfoString()

Obtain string-valued information relating to the current message part.

Syntax

```
const char *mtaDecodeMessageInfoString(mta_decode_t *dctx,
                                       int           item,
                                       const char  **str,
                                       size_t       *len);
```

Arguments

Table 6–44 lists the **mtaDecodeMessageInfoString()** arguments.

Table 6–44 mtaDecodeMessageInfoString() Arguments

Arguments	Description
dctx	A decode context created by **mtaMessageDecode()**.
item	Item identifier specifying which string-value item to return. See the description that follows for the list of permitted values for this argument.
str	An optional pointer to receive the address of the requested string. The string will be NULL terminated. A value of NULL may be passed for this argument.
len	An optional pointer to receive the length of the requested string. This length is measured in bytes and does not include the NULL terminator at the end of the string. A value of NULL may be passed for this argument.

Description

This routine is used to obtain string-valued information about the current message part. (When **mtaDecodeMessage()** calls either a user-supplied inspection or output routine, it provides a decode context describing the current message part being processed.)

Table 6–45 lists the values for the **item** call argument, and gives a description of each.

Table 6–45 mtaDecodeMessageInfoString() Item Call Argument Values

Values	Description
MTA_DECODE_ CCHARSET	The character set specified with the **CHARSET** parameter of the part's **Content-type:** header line. If the part lacks a **CHARSET** specification, then the value **us-ascii** will be returned.
MTA_DECODE_CDISP	Value of the **Content-disposition:** header line, less any optional parameters. If the part lacks a **Content-disposition:** header line, the returned value will be a zero length string.
MTA_DECODE_ CSUBTYPE	The content subtype specified with the part's **Content-type:** header line (for example, **plain** for **text/plain**, **gif** for **image/gif**). Defaults to **plain** when the part lacks a **Content-type:** header line.
MTA_DECODE_CTYPE	The major content type specified with the part's **Content-type:** header line (for example, **text** for **text/plain**, **image** for **image/gif**). Defaults to **text** when the part lacks a **Content-type:** header line.

Return Values

mtaDecodeMessageInfoString() always returns a value for the **CHARSET** parameter of the **Content-type:** header line. When the **Content-type:** header line is not present, it returns the MIME default value, **us-ascii**.

Upon normal, successful completion a pointer to the requested string is returned. In addition, if pointers were provided in the **str** and **len** call arguments, the address of the string and its length are returned.

In the event of an error, a NULL value is returned and **mta_errno** is set to indicate the error status code. Table 6–46 lists the error status codes, and gives a description of each.

Table 6–46 mtaDecodeMessageInfoString() Error Status Codes

Error Status Codes	Description
MTA_BADARGS	A NULL value was supplied for the **dctx** call argument, or an invalid decode context was supplied for **dctx**.
MTA_NOSUCHITEM	An invalid value was supplied for the **item** call argument.

Example

```
printf("The message part&rsquo;s character set is %s\n",
       mtaDecodeMessageInfoString(dctx, MTA_DECODE_CCHARSET,
                                  NULL, NULL));
```

mtaDecodeMessagePartCopy()

Explicitly copy a message part to the message being written.

Syntax

```
int mtaDecodeMessagePartCopy(mta_decode_t *dctx,
                             int           item_code, ...);
```

Arguments

Table 6–47 lists the **mtaDecodeMessagePartCopy()** arguments.

Table 6–47 mtaDecodeMessagePartCopy() Arguments

Arguments	Description
dctx	A decode context created by **mtaMessageDecode()**.
item_code	Reserved for future use. A value of zero must be supplied for this argument.

Description

When an output routine is used in conjunction with mtaDecodeMessage(), the inspection routine can explicitly request that the current message part be copied to the output destination. After the inspection routine calls **mtaDecodeMessagePartCopy()**, it will no longer be called for that message part.

If the inspection routine is called with a data type of **MTA_DATA_NONE**, the message part copy is implicitly done, even if the inspection routine does not call either **mtaDecodeMessagePartCopy()** or mtaDecodeMessagePartDelete(). Therefore, the only advantage to making an explicit call to **mtaDecodeMessagePartCopy()** is that once that call is made, the inspection routine will no longer be called for that particular message part.

Return Values

Table 6–48 lists the **mtaDecodeMessagePartCopy()** return values.

Table 6–48 mtaDecodeMessagePartCopy() Return Values

Values	Description
0	Normal, successful completion.
MTA_BADARGS	A NULL value was supplied for the **dctx** call argument, or an invalid decode context was supplied for **dctx**.
MTA_NO	Invalid call to this routine. Either no output routine is being used, or the call was made from the output routine itself.Output errors encountered while attempting to write the output may also result in this error.

Example

This routine is used in "Example 5-2 Decoding MIME Messages Complex Example".

mtaDecodeMessagePartDelete()

Prevent a message part from being written or replace it with a text part.

Syntax

```
int mtaDecodeMessagePartDelete(mta_decode_t *dctx,
                               int           item_code, ...);
```

Arguments

Table 6–49 lists the **mtaDecodeMessagePartDelete** arguments.

Table 6–49 mtaDecodeMessagePartDelete Arguments

Arguments	Description
dctx	A decode context created by **mtaMessageDecode()**.
item_code	An optional list of item codes. See the description section that follows for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

When an output routine is used in conjunction with mtaDecodeMessage(), the inspection routine may discard the current message part by calling this routine. As an alternative to discarding the part, it may be replaced with a part containing caller-supplied data such as a warning message. This replacement is achieved through the use of item codes.

Once **mtaDecodeMessagePartDelete()** has been called, the inspection routine will no longer be called for that message part. As such, calling the routine is final and cannot be undone short of cancelling the entire message decode operation itself (for example, by having the caller-supplied read routine return an error, or after **mtaDecodeMessage()** completes, cancelling the dequeue and enqueue operations with **mtaDequeueMessageFinish()** and **mtaEnqueueFinish()**).

Table 6–50 lists the item codes for this routine, any additional item code arguments each item code requires, and gives a description of each.

Table 6–50 mtaDecodeMessagePartDelete Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ DECODE_ CCHARSET	**const char *charsetsize_t charset_len**	Specify the character set used for the message part (for example, **us-ascii**, **iso-8859-1**). This item code must be followed by two additional call arguments:# The name of the character set # The length in bytes of that name If a value of zero is passed for the length, then the name must be NULL terminated. |
MTA_ DECODE_ CDISP	**const char *dispositionsize_ t disposition_len**	Specify the content disposition for the message part (for example, **inline**, **attachment**; **filename=a.doc**). This disposition information will be placed in a **Content-disposition:** header line. The item code must be followed by two additional call arguments:# The disposition string # The length in bytes of that string If a value of zero is passed for the length, then the disposition string must be NULL terminated. |
MTA_ DECODE_ CLANG,	**const char *languagesize_t language_len**	Specify the language used for the message part (for example, **en**, **fr**). This language information will be placed in a **Content-language:** header line. The item code must be followed by two additional call arguments:# The language string # The length in bytes of that string. If a value of zero is passed for the length, then the string must be NULL terminated. |

Table 6–50 (Cont.) mtaDecodeMessagePartDelete Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ DECODE_ CSUBTYPE	**const char *subtypesize_t subtype_len**	Specify the content subtype for the message part (for example, **plain** or **html** for **text/plain** or **text/html**). This subtype information will be combined with the **type** and **charset** information and placed in a **Content-type:** header line. The item code must be followed by two additional call arguments:# The language string # The length in bytes of that string. If a value of zero is passed for the length, then the string must be NULL terminated. |
MTA_ DECODE_ CTYPE	**const char *typesize_t type_ len**	Specify the major content type for the message part (for example, **text** for **text/plain** or **text/html**). This major type information will be combined with the subtype and charset information and placed in a **Content-type:** header line. The item code must be followed by two additional call arguments:# The language string # The length in bytes of that string. If a value of zero is passed for the length, then the string must be NULL terminated. |
MTA_ITEM_ LIST	**mta_item_list_t *item_list**	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of 0. For further information on item lists, see "Item Codes and Item Lists".
MTA_ REASON	**const char *textsize_t text_ len**	Specifies the content and length of caller-supplied text or data used to replace the deleted message part.The item code must be followed by two additional call arguments:# The language string # The length in bytes of that string. If a value of zero is passed for the length, then the string must be NULL terminated. |

Return Values

Table 6–51 lists the **mtaDecodeMessagePartDelete** return values.

Table 6–51 mtaDecodeMessagePartDelete Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	Returned for one of two reasons:# A NULL value was supplied for the **dctx** call argument, an invalid decode context was supplied for **dctx**. # A required argument to an item code was NULL. |
MTA_NO	Returned for one of two reasons:# Invalid call. Either no output routine is being used, or the call was made from the output routine itself. # Output errors encountered while attempting to write the output. |
MTA_NOSUCHITEM	An invalid item code was specified.

Example

The following code fragment shows how the routine is used to discard the message part:

```
mtaDecodeMessagePartDelete(dctx, 0);
```

The following code fragment shows how to replace the message part with a text warning:

```
mtaDecodeMessagePartDelete(dctx,
    MTA_REASON, "Warning: virus infected message part was
                 discarded.", 0,&rdquo;
    MTA_DECODE_CLANG, "en", 2,
    MTA_DECODE_CCHARSET, "us-ascii", 8, 0);
```

The following code fragment shows the output generated by the preceding code example.

```
Content-type: text/plain; charset=us-ascii
Content-language: en

Warning: virus infected message part was discarded.
```

See also "Example 5-2 Decoding MIME Messages Complex Example".

mtaDequeueInfo()

Obtain information associated with an ongoing message dequeue.

Syntax

```
int mtaDequeueInfo(mta_dq_t *dq_ctx,
                   int       item_code, ...);
```

Arguments

Table 6–52 lists the **mtaDequeueInfo()** arguments.

Table 6–52 mtaDequeueInfo() Arguments

Arguments	Description
dq_ctx	A dequeue context created by **mtaDequeueStart()**.
item_code	An optional list of item codes. See the description section that follows for a list of item codes. The list must be terminated with an integer argument with value 0.

Description

Information associated with an ongoing message dequeue may be obtained with **mtaDequeueInfo()**. The information to obtain is specified through the use of item codes, as show in Table 6–53.

Note: The pointers returned by **mtaDequeueInfo()** are only valid during the life of the dequeue context. Once the dequeue has been completed for that particular message, the pointers are no longer valid.

Table 6–53 mtaDequeueInfo Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ CHANNEL	**const char **channelsize_t *channel_len**	Obtain the name of the channel for which messages are being dequeued. The channel name will be NULL terminated.This item code must be followed by two additional call arguments:# the address of a pointer to receive the address of the NULL terminated channel name. # The address of a **size_t** to receive the length of the channel name. A NULL value may be passed for the **channel_len** argument. |
MTA_ DELIVERY_ FLAGS	**size_t *dflags**	Return the envelope delivery flags for either the entire message or for a particular recipient. If called before the first call to **mtaDequeueRecipientNext()**, then the delivery flags for the entire message are returned. If called after the first call to **mtaDequeueRecipientNext()**, then the delivery flags are returned for the most recently reported envelope recipient address. The value of the delivery flags is a logical OR of the **deliveryflags** channel option values on each channel the message has been enqueued to as it flows through the MTA.This item code must be followed by one additional call argument, the address of a **size_t** to receive the delivery flag setting.
MTA_ DOMAIN	**const char **domainsize_t *domain_len**	Retrieve the destination domain name, if any, the Job Controller has associated with this dequeue thread. When the channel is marked with the **single_sys** channel option, then the Job Controller tries to give each dequeue thread for that channel all messages destined for the same host as determined by the domain name in the recipient envelope addresses.This item code must be followed by two additional call arguments:# The address of a pointer to receive the address of the NULL terminated destination domain name. # The address of a **size_t** to receive the length of that domain name. A NULL value may be passed for the **domain_len** argument. |
MTA_ENV_ ID	**const char **env_ idsize_t *env_id_ len**	Obtain the envelope ID associated with this message. If the message was submitted to the MTA using the SMTP NOTARY extension (RFC 1891), then this will be the value of the **ENVID** parameter supplied with the SMTP **MAIL FROM** command. In all other cases, it will be an envelope ID assigned by the MTA.This item code must be followed by two additional call arguments:# The address of a pointer to receive the address of the NULL terminated envelope ID. # The address of a **size_t** to receive the length of that envelope id. A NULL value may be passed for the **env_id_len** argument. |
MTA_ENV_ TO	**const char **env_ tosize_t *env_to_ len**	Return the envelope recipient address last returned by **mtaDequeueRecipientNext()**. If that routine has not yet been called for the dequeue context, then an **MTA_NO** error code will be returned.This item code must be followed by two additional call arguments:# The address of a pointer to receive the address of the NULL terminated recipient address. # The address of a **size_t** to receive the length of that address. A NULL value can be passed for the **env_to_len** argument. |

Table 6–53 (Cont.) mtaDequeueInfo Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_ENV_ FROM	**const char **env_ fromsize_t *env_ from_len**	Obtain the envelope **From:** address for the message. It is possible for this to be an empty string (that is, a string of zero length). This is not uncommon and is mandated by Internet standards for automatically generated notification addresses. Notifications must never be sent for messages with an empty envelope **From:** address. The MTA SDK adheres to this rule when generating any requested notification messages.This item code must be followed by two additional call arguments:# The address of a pointer to receive the address of the NULL terminated envelope **From:** address. # The address of a **size_t** to receive the length of that address. A NULL value can be passed for the **env_from_len** argument. |
MTA_ IRCPT_TO	**const char **ircpt_tosize_t *ircpt_to_len**	Return the intermediate form of the last envelope recipient address returned by **mtaDequeueRecipientNext()**. If that routine has not yet been called for the dequeue context, then an **MTA_NO** error code will be returned.This item code must be followed by two additional call arguments:# The address of a pointer to receive the address of the NULL terminated intermediate recipient address # The address of a **size_t** to receive the length of that address. A NULL value can be passed for the **ircpt_to_len** argument. |
MTA_ITEM_ LIST	**mta_item_list_ t*item_list**	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of zero. For further information on item list usage, see "Item Codes and Item Lists".

Return Values

Table 6–54 lists the **mtaDequeueInfo** return values.

Table 6–54 mtaDequeueInfo Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	Received for one of three reasons:# A NULL value was supplied for the **dq_ctx** call argument # An invalid dequeue context was supplied for **dq_ctx** # A required argument to an item code was NULL. |
MTA_NO	An attempt was made to retrieve recipient information before calling **mtaDequeueRecipientNext()**.
MTA_NOSUCHITEM	An invalid item code was specified.
MTA_THREAD	The MTA SDK detected simultaneous use of the dequeue context by two different threads.

Example

The following code fragment illustrates how this routine is used to retrieve the delivery flags and intermediate recipient address for each recipient address.

```
int dflags, istat;
const char *to, *ito;

while (!(istat = mtaDequeueRecipientNext(dq, &amp;to, NULL, 0)))
```

```
{
    mtaDequeueInfo(dq, MTA_DELIVERY_FLAGS, &amp;dflags,
                   MTA_IRCPT_TO, &amp;ito, NULL, 0);
    printf("Delivery flags: %d\n"
           "Intermediate recipient address: %s\n", dflags, ito);
}
if (istat != MTA_EOF)
    printf("An error occured; %s\n", mtaStrError(istat));
```

mtaDequeueLineNext()

Read the next line of the message from the queued message file.

Syntax

```
int mtaDequeueLineNext(mta_dq_t    *dq_ctx,
                       const char **line,
                       size_t      *len);
```

Arguments

Table 6–55 lists the **mtaDequeueLineNext()** arguments.

Table 6–55 mtaDequeueLineNext() Arguments

Arguments	Description
dq_ctx	A dequeue context created by **mtaDequeueStart()**.
line	Optional address of a pointer to receive the address of the next line of the message. The line will not be NULL terminated. A value of NULL may be passed for this argument.
len	Optional address of a **size_t** to receive the length of the returned line. A value of NULL may be passed for this argument.

Description

After exhausting a message's list of envelope recipients by repeated calls to **mtaDequeueRecipientNext()**, begin reading the message's header and content with **mtaDequeueLineNext()**. Each call will return one line of the message, with the first call returning the first line of the message, the second call the second line, and so on. Once the message has been completely read, the status code **MTA_EOF** will be returned.

The returned lines of the message will not be NULL terminated. This is because the underlying message file is often mapped into memory. When that is the case, then the returned pointer is a pointer into that memory map. Since the message files themselves do not contain NULL terminators and the file is mapped read-only, it is not possible for the SDK to add a NULL terminator to the end of the line without copying it first to a writable portion of memory.

The returned lines of the message will not have any line terminators such as a line feed or a carriage return. It is up to the calling routine to supply whatever line terminators might be appropriate (for example, adding a carriage-return line-feed pair when transmitting the line over SMTP.)

It is possible to call **mtaDequeueLineNext()** with NULL values for both the **line** and **len** call arguments. But this is of limited use; one example is when writing a channel that deletes all queued messages after first counting the number of lines in each

message for accounting purposes. More typical of such a channel would be to supply NULL for the **line** argument but pass a non-zero address for the **len** argument. That would then allow the channel to count up the number of bytes in the deleted message.

Return Values

Table 6–56 lists the **mtaDequeueLineNext()** return values.

Table 6–56 mtaDequeueLineNext() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	A NULL value was supplied for the **dq_ctx** call argument, or an invalid dequeue context was supplied for **dq_ctx**.
MTA_EOF	Message file has been completely read; no further lines to return.

Example

```
int istat;
const char *line;
size_t len;

while (!(istat = mtaDequeueLineNext(dq_ctx, &amp;line, &amp;len)))
     printf("%.*s\n", len, line);
if (istat != MTA_EOF)
     printf("An error occured; %s\n", mtaStrError(istat));
```

mtaDequeueMessageFinish()

Complete a message dequeue or defer a message for later processing.

Syntax

```
int mtaDequeueMessageFinish(mta_dq_t *dq_ctx,
                            int       item_code, ...);
```

Arguments

Table 6–57 lists the **mtaDequeueMessageFinish()** arguments.

Table 6–57 mtaDequeueMessageFinish() Arguments

Arguments	Description
dq_ctx	A dequeue context created by **mtaDequeueStart()**.
item_code	An optional list of item codes. See the description section the follows for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

Before completing processing of a queued message, the disposition of each envelope recipient must be set either by repeated calls to **mtaDequeueRecipientDisposition()**, or by means of the **MTA_DISP** item code for **mtaDequeueMessageFinish()**. For the former, a call should be made for each envelope recipient address. For the latter, the disposition set with **MTA_DISP** applies to all envelope recipients, overriding any previous settings made with **mtaDequeueRecipientDisposition()**. It is important that

the dispositions be set correctly because they influence whether or not the message is deleted from the channel's queue by **mtaDequeueMessageFinish()**. Incorrectly setting the dispositions can lead to duplicate message delivery, or, worse yet, lost mail.

To complete processing of a queued message, call **mtaDequeueMessageFinish()**. Upon being called, the routine performs one of three possible actions:

- If all recipients have a disposition indicating successful processing or a permanent failure, then the underlying message file is deleted from the channel's queue and any necessary notification messages are sent. This corresponds to the dispositions: **MTA_DISP_DELIVERED**, **MTA_DISP_FAILED**, **MTA_DISP_RELAYED**, **MTA_DISP_RELAYED_FOREIGN**, **MTA_DISP_RETURN**, and **MTA_DISP_TIMEDOUT**.
- If all recipients have a disposition indicating a temporary processing problem or if the **MTA_ABORT** item code is specified, then the message file is left in the channel's queue and a subsequent processing attempt is scheduled. The **MTA_DISP_DEFERRED** disposition is the only disposition that indicates a temporary processing problem. Generation of delay notifications is handled by a special MTA process referred to as the return job. Generation of delay notifications is not handled by **mtaDequeueMessageFinish()**.
- If only a subset of the recipients have a disposition indicating a temporary processing problem, then a new message is placed in the channel's queue. This new message is identical to the current message being processed except that its envelope recipient list contains just those recipients whose disposition indicates a temporary processing problem. The current message being processed is then removed from the channel's queue and any necessary notifications are sent for the recipients that had dispositions indicating successful processing or a permanent failure.

After **mtaDequeueMessageFinish()** is called, the dequeue context passed to it is no longer valid, regardless of the status it returns. When it returns an error status, it also defers the message and all of its recipients for later processing. This is done regardless of the disposition of the recipients. Doing otherwise could potentially lead to lost mail.

Internet standards require that notifications concerning a message be directed to the message's envelope **From:** address. In addition, the following two rules apply:

- Automatically generated notification messages themselves must have an empty envelope **From:** address.
- Notifications must not be sent for messages with an empty envelope **From:** address.

These two rules combine to prevent certain broad classes of message loops. The MTA SDK strictly adheres to these Internet requirements.

Whenever a temporary processing error occurs and the channel can no longer process a queued message, processing of the message should be deferred until a later time. Processing for all recipients is deferred regardless of any prior disposition settings. Temporary processing errors include such errors as: insufficient virtual memory, network problems, disk errors, and other unexpected processing errors.

Table 6–58 lists the item codes for this routine, the additional arguments they take, and gives a description of each one.

Table 6–58 mtaDequeueMessageFinish() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ABORT	None	When this item code is specified, processing of the message is deferred for all recipients of the message. The message is left in the channel's queue and a later processing attempt is scheduled.
MTA_DISP	**size_t disposition**	Use the **MTA_DISP** item code to set the disposition for all recipients of the message. This disposition will override any prior disposition settings.This item code must be followed by one additional call argument: the disposition value to set. See the description of **mtaDequeueRecipientDisposition()** for a discussion of the disposition settings.
MTA_ITEM_ LIST	mta_item_list_t *item_list	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of zero. For further information on item list usage, see "Item Codes and Item Lists".
MTA_REASON	**const char *reasonsize_t reason_len**	When deferring processing of a message, the reason for the deferral may be saved as part of the messages delivery history. This delivery history may be viewed by system managers with the MTA **qm** utility. It may also be reported in delay notifications.This item code must be followed by two additional call arguments:# The address of the string containing the reason text. # The length in bytes of the reason text. If a value of zero is passed for the length, then the reason text must be NULL terminated. |

Return Values

Table 6–59 lists the **mtaDequeueMessageFinish()** return values.

Table 6–59 mtaDequeueMessageFinish() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	Received for one of two reasons:# A NULL value was supplied for the **dq_ctx** call argument, an invalid dequeue context was supplied for **dq_ctx**. # A required argument to an item code was NULL. |
MTA_NO	Unable to dequeue the message. This error can result from an attempt to enqueue a new message to a subset of recipients.
MTA_NOSUCHITEM	An invalid item code was specified.
MTA_ORDER	Call made out of sequence. The call was made either before the recipient list has been exhausted with **mtaDequeueRecipientNext()**, or after the message had been dequeued or deferred with **mtaDequeueMessageFinish()**.
MTA_THREAD	The MTA SDK detected simultaneous use of the dequeue context by two different threads.

Example

There are three code examples, each showing variations on deferring a message.

The following code fragment shows how to use this routine to defer processing of a message until a later time by calling the routine with the **MTA_ABORT** item code:

```
mtaDequeueMessageFinish(dq_ctx, MTA_ABORT, 0);
```

The following code fragment shows how to use this routine to defer processing of a message and setting the disposition:

```
mtaDequeueMessageFinish(dq_ctx, MTA_DISP, MTA_DISP_DEFERRED, 0);
```

The following code fragment shows how to use this routine to defer processing of a message with a text string explaining the reason for the deferral:

```
mtaDequeueMessageFinish(dq_ctx, MTA_ABORT, MTA_REASON,
                        "Temporary network error", 0, 0);
```

mtaDequeueRecipientDisposition()

Specify the delivery status (disposition) of an envelope recipient address.

Syntax

```
int mtaDequeueRecipientDisposition(mta_dq_t   *dq_ctx,
                                   const char *env_to,
                                   size_t      env_to_len,
                                   size_t      disposition,
                                   int         item_code, ...);
```

Arguments

Table 6–60 lists the mtaDequeueRecipientDisposition() arguments.

Table 6–60 mtaDequeueRecipientDisposition() Arguments

Arguments	Description
dq_ctx	A dequeue context created by **mtaDequeueStart()**.
env_to	The recipient address to effect the setting for. This must be the recipient's envelope **To:** address as returned by **mtaDequeueRecipientNext()** and not some transformation of that address. If a value of zero is passed for the **env_to_len** argument, then this string must be NULL terminated.
env_to_len	The length in bytes of the recipient address, **env_to**. This length does not include any NULL terminator. If a value of zero is passed for this argument, then the recipient address string must be NULL terminated.
disposition	The delivery status disposition to set for this recipient address. See the description section that follows for further details.
item_code	An optional list of item codes. See the description section that follows for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

Before completing processing of a queued message, the disposition of each envelope recipient must be set either by repeated calls to **mtaDequeueRecipientDisposition()**,

or by means of the **MTA_DISP** item code for **mtaDequeueMessageFinish()**. For the former, a call should be made for each envelope recipient address. For the latter, the disposition set with **MTA_DISP** applies to all envelope recipients, overriding any previous settings made with **mtaDequeueRecipientDisposition()**. The delivery status dispositions, and their descriptions are listed in Table 6–61. Pass one of these values for the disposition argument.

Table 6–61 mtaDequeueRecipientDisposition() Delivery Status Dispositions

Delivery Status Dispositions	Description
MTA_DISP_DEFERRED	Processing for this recipient has experienced a temporary failure (for example, the network is temporarily down, the disk is currently full, the recipient is presently over quota). Schedule a later processing attempt for this recipient.
MTA_DISP_ DELIVERED	Final delivery has been effected for this recipient address. Any required delivery notifications should be generated. Intermediate processing channels should use **MTA_DISP_RELAYED** rather than **MTA_DISP_DELIVERED**. Use of **MTA_DISP_ DELIVERED** by an intermediate processing channel might incorrectly generate a delivery status notification when final delivery has not yet been effected.
MTA_DISP_FAILED	Processing for this recipient has experienced a permanent failure. The message is and will remain undeliverable for this recipient. No further delivery attempts are to be made for this recipient. Any required non-delivery notifications should be generated.
MTA_DISP_RELAYED	The message has been successfully processed for this recipient. No further processing by this channel is needed for this recipient address. No delivery status notification is generated as final delivery will be effected by another entity capable of generating any needed notification messages. This disposition should be used by intermediate processing channels. It should also be used by gateways that transfer the message to other mail systems capable of generating the necessary notification messages.
MTA_DISP_RELAYED_ FOREIGN	The message has been successfully processed for this recipient. No further processing by this channel is needed for this recipient address; however, a relayed delivery status notification should be generated if delivery notification was requested for this recipient. This disposition should be used by gateways that transfer the message to other mail systems incapable of generating the necessary notification messages.
MTA_DISP_RETURN	Generate a postmaster non-delivery notification for this recipient and, for this recipient, remove the message from the channel's queue. This disposition is not intended for use by channels. Instead, it should be used by postmaster utilities that allow the postmaster to manually return mail messages.
MTA_DISP_TIMEDOUT	Generate a timed-out non-delivery notification indicating that the message has been undeliverable for too long and no further delivery attempts will be made. This disposition is not intended for use by channels. Instead, it is meant for use by the MTA return job that scans the MTA queues, returning old, undeliverable messages to their originators.

Table 6–62 lists the item codes for this routine, and the additional required arguments, and gives a description of each.

Table 6–62 mtaDequeueRecipientDisposition() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_DISP	**size_t disposition**	Use the **MTA_DISP** item code to set the disposition for all recipients of the message. This disposition will override any prior disposition settings. This item code must be followed by one additional call argument: the disposition value to set. See the description of **mtaDequeueRecipientDisposition()** for a discussion of the disposition settings.
MTA_ ITEM_LIST	**mta_item_list_t *item_list**	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of zero. For further information on item list usage, see "Item Codes and Item Lists".
MTA_ REASON	**const char *reasonsize_t reason_len**	The reason for ascribing the disposition to this recipient address. This reason might then appear in any delivery or non-delivery status notification for that recipient.This item code must be followed by two additional call arguments:# the address of the string containing the reason text. # The length in bytes of the reason text. If a value of zero is passed for the length, then the reason text must be NULL terminated. |

Return Values

Table 6–63 lists the **mtaDequeueRecipientDisposition()** return values.

Table 6–63 mtaDequeueRecipientDisposition() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value was returned for one of the following reasons:# A NULL value was supplied for the **dq_ctx** call argument. # An invalid dequeue context was supplied for **dq_ctx**. # A required argument to an item code was NULL. |
MTA_NOSUCHITEM	An invalid item code was specified.
MTA_THREAD	The MTA SDK detected simultaneous use of the dequeue context by two different threads.

Example

This code fragment assumes a condition in which the recipient address is invalid. It returns a disposition of **MTA_DISP_FAILED** with an explanation.

```
mtaDequeueRecipientDisposition(
   dq_ctx, "sue@siroe.com", 0, MTA_DISP_FAILED,
   MTA_REASON, "Invalid recipient address: no such user", 0, 0);
```

mtaDequeueRecipientNext()

Obtain the next envelope recipient address for the queued message file.

Syntax

```
int mtaDequeueRecipientNext(mta_dq_t    *dq_ctx,
                            const char **env_to,
```

```
size_t      *env_to_len,
int          item code, ...);
```

Arguments

Table 6–64 lists the mtaDequeueRecipientNext() arguments.

Table 6–64 mtaDequeueRecipientNext() Arguments

Argument	Description
dq_ctx	A dequeue context created by **mtaDequeueStart()**.
env_to	Optional address of a pointer to receive the memory address of the next envelope recipient address. The recipient address will not be NULL terminated.
env_to_len	Optional address of a **size_t** to receive the length of the returned recipient address. A value of NULL may be passed for this argument.
item_code	An optional list of item codes. See the description section that follows for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

The first step in processing a queued message is to retrieve its list of envelope recipient addresses. This is done by repeatedly calling **mtaDequeueRecipientNext()** until a status code of **MTA_EOF** is returned. Note that each call that returns a recipient address will return a status code of **0** (**MTA_OK**). The final call, which returns **MTA_EOF**, will not return a recipient address.

The processing of the list of envelope recipient addresses will, in general, be unique to each channel. Intermediate processing channels should simply re-enqueue a new message and copy the envelope recipient list verbatim over to the new message being enqueued, being sure to specify the **MTA_ENV_TO** and **MTA_DQ_CONTEXT** item codes to **mtaEnqueueTo()**. The envelope recipient list must be read in its entirety before attempting to read the message itself with **mtaDequeueLineNext()**. Failure to do so will result in an **MTA_ORDER** error being returned by **mtaDequeueLineNext()**.

This routine accepts the same item codes as mtaDequeueInfo(). The code fragments are equivalent also, (compare the examples). Consequently, the **mtaDequeueInfo()** routine might appear superfluous. However, it also serves as a means of obtaining, in a single, non-repeated call, information about the overall message itself, such as the message's envelope ID.

Return Values

Table 6–65 lists the **mtaDequeueRecipientNext()** return values.

Table 6–65 mtaDequeueRecipientNext() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value was returned for one of the following reasons:# A NULL value was supplied for the **dq_ctx** call argument. # An invalid dequeue context was supplied for **dq_ctx**. # A NULL value was supplied for a required item code argument. |
MTA_NOMEM	Insufficient virtual memory.

Table 6–65 (Cont.) mtaDequeueRecipientNext() Return Values

Return Values	Description
MTA_EOF	The recipient list has been completely read; there are no further recipient addresses to return.
MTA_THREAD	Concurrent use of the dequeue context by two different threads has been detected.

Example

This code fragment illustrates an intermediate processing channel using this routine to fetch recipient addresses.

```
int dflags, istat;
const char *to, *ito;
while (!(istat = mtaDequeueRecipientNext(dq, &amp;to, NULL,
                                MTA_DELIVERY_FLAGS, &amp;dflags,
                                MTA_IRCPT_TO, &amp;ito, NULL, 0)))
    printf("Delivery flags: %d\n"
           "Intermediate recipient address: %s\n", dflags, ito);
if (istat != MTA_EOF)
    printf("An error occured; %s\n", mtaStrError(istat));
```

mtaDequeueRewind()

Reset the read point to the start of the message.

Syntax

```
int mtaDequeueLineNext(mta_dq_t *dq_ctx);
```

Arguments

Table 6–66 lists the **mtaDequeueRewind()** arguments.

Table 6–66 mtaDequeueRewind() Arguments

Arguments	Description
dq_ctx	A dequeue context created by **mtaDequeueStart()**.

Description

Repositions the read point back to the start of the message.

After obtaining a message's recipient list by repeated calls to **mtaDequeueRecipientNext()**, the read point into the underlying message file is positioned at the start of the actual message. Specifically, at the start of the message's outermost header. Calling **mtaDequeueLineNext()** advances this read point, with each call moving it towards the end of the message. To reposition the read point back to the start of the message (that is, to the start of the message's outermost header), call **mtaDequeueRewind()**. Use this call if a program needs to make a second pass through a message. For example, a program might scan a message's content before actually processing it.

Return Values

Table 6–67 lists the **mtaDequeueRewind()** return values.

Table 6–67 mtaDequeueRewind() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	A NULL value was supplied for the **dq_ctx** call argument, or an invalid dequeue context was supplied for **dq_ctx**.
MTA_ORDER	Call made out of sequence. The call was made either before the recipient list has been exhausted with **mtaDequeueRecipientNext()**, or after the message had been dequeued or deferred with **mtaDequeueMessageFinish()**.
MTA_THREAD	The MTA SDK detected simultaneous use of the dequeue context by two different threads.

Example

None

mtaDequeueStart()

Initiate message dequeue processing.

Syntax

```
int mtaDequeueStart(void                     *ctx1,
                    mta_dq_process_message_t *process_message,
                    mta_dq_process_done_t    *process_done,
                    int                       item_code, ...);
```

Arguments

Table 6–68 lists the **mtaDequeueStart()** arguments.

Table 6–68 mtaDequeueStart() Arguments

Arguments	Description
ctx1	Optional pointer to a caller-supplied context or other data type. This pointer will be passed as the **ctx1** argument to the caller-supplied routines **process_message** and **process_done**. A value of NULL may be passed for this argument.
process_message	The address of a caller-supplied routine to process each message.
process_done	Optional address of a caller-supplied clean up routine. A NULL value may be passed for this argument.
item_code	An optional list of item codes. See the description section that follow for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

The **mtaDequeueStart()** routine initiates processing of messages queued to a specific channel. By default, the channel serviced will be determined from the **PMDF_CHANNEL** environment variable. However, a channel name can be explicitly specified with the **MTA_CHANNEL** item code.

All of the item codes, their additional arguments, and a description of each are included in Table 6–69.

Table 6–69 mtaDequeueStart() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ CHANNEL	**const char *channelsize_t channel_len**	Explicitly specify the name of the channel name to perform dequeue processing for. This item code must be followed by two additional call arguments: the name of the channel and the length in bytes of that channel name. If a value of zero is passed for the length, then the channel name must be NULL terminated.When this item code is not specified, the name of the channel to process queued messages for is taken from the **PMDF_CHANNEL** environment variable.
MTA_ITEM_ LIST	**mta_item_list_t *item_list**	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of zero. For further information on item list usage, see "Item Codes and Item Lists".
MTA_JBC_ MAX_ ATTEMPTS	**size_t attempts**	Specify the maximum number of contiguous attempts that will be made to sleep and then re-query the Job Controller for work after being told by the Job Controller that there are no more messages to process. The default value for this setting is **5** attempts. If an attempt succeeds in providing additional work, the count of attempts is reset to zero. (The duration of each sleep may be specified with the **MTA_JBC_RETRY_INTERVAL** item code.)This item code must be followed by an additional argument: the maximum number of contiguous attempts to make.
MTA_JBC_ RETRY_ INTERVAL	**size_t seconds**	Set the number of seconds **mtaDequeueMessage()** sleeps before again querying the Job Controller for additional work. When not specified, a value of **10** seconds is used. This item code must be followed by one additional argument: the number of seconds to sleep for.
MTA_ THREAD_ MAX_ THREADS	**size_t threads**	Specify the maximum number of processing threads to run concurrently. If not specified, then a limit of **20** threads is assumed.This item code must be followed by one additional argument: the maximum number of concurrent threads to permit.
MTA_ THREAD_ STACK_SIZE	**size_t bytes**	By default, the processing threads will have a stack whose size is sufficient for MTA SDK operations. This is the size returned by the **mtaStackSize()** routine. To request a larger size, use this item code to specify the desired size. Note that specification of a smaller size is ignored: **mtaDequeueMessage()** will never use a stack size smaller than that returned by **mtaStackSize()**.This item code must be followed by one additional argument: the minimum size in bytes for each thread's stack.
MTA_ THREAD_ MAX_ MESSAGES	**size_t messages**	The number of messages to allocate per processing thread. The channel program will aim to run N processing threads where N is computed as follows: N = (count of pending queued messages) / **MTA_THREAD_MAX_ MESSAGES.** For example, if there are **100** queued messages and **MTA_THREAD_MAX_MESSAGES** has its default value of **20** messages, then **5** processing threads are started.This value does not control the total number of messages presented to a single processing thread.This item code must be followed by one additional argument: the number of messages for each processing thread.

Table 6–69 (Cont.) mtaDequeueStart() Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_ THREAD_ WAIT_ TIMEOUT	**size_t seconds**	Once **mtaDequeueMessage()** determines that there are no more messages to process, it waits for all processing threads to complete their work and exit. By default, **mtaDequeueMessage()** will wait no longer than 1800 seconds (30 minutes).This item code must be followed by one additional argument: the maximum number of seconds to wait.

Return Values

Table 6–70 lists the **mtaDequeueStart()** return values.

Table 6–70 mtaDequeueStart() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of following reasons:# A NULL value was supplied for the **dq_ctx** call argument. # An invalid dequeue context was supplied for **dq_ctx**. # A NULL value was specified for the **process_message** routine. # A NULL value was supplied for a required item code argument. |
MTA_FOPEN	Unable to initialize the MTA SDK. Unable to read one or more configuration files.For further information, issue the following command: **imsimta test -rewrite**
MTA_NETWORK	Error communicating with the Job Controller.
MTA_NO	Unable to initialize the MTA SDK.For further information, issue the following command: **imsimta test -rewrite**
MTA_NOMEM	Insufficient virtual memory.
MTA_NOSUCHCHAN	Specified channel is not defined in the MTA configuration file. If no channel was explicitly specified, then the channel name specified with the **PMDF_CHANNEL** environment variable is not defined in the MTA configuration file. This error may also be returned when the Job Controller's configuration file lacks a **CHANNEL** section matching the specified channel.
MTA_NOSUCHITEM	An invalid item code was specified.

Example

For an example of **mtaDequeueStart()**, see "Example 5-2 Decoding MIME Messages Complex Example".

Other Considerations for mtaDequeueStart()

This section contains supplementary information concerning **mtaDequeueStart()**. It covers the following topics:

- Multiple Calls to mtaDequeueStart()
- Message Processing
- Message Processing Procedure
- process_message() Routine

- process_done() Routine
- Thread Creation Loop

Multiple Calls to mtaDequeueStart()

A channel program can call **mtaDequeueStart()** multiple times: either sequentially or in parallel. In the latter case, the program would need to create threads so as to effect multiple, simultaneous calls to **mtaDequeueStart()**. However, just because this can be done does not mean that it is appropriate to do so. In the former case of multiple sequential calls, there's no need to be making repeated calls. When **mtaDequeueStart()** returns, the channel no longer needs immediate processing and has been in that state for

MTA_JBC_ATTEMPTS_MAX * MTA_JBC_RETRY_INTERVAL

seconds. Instead, the channel program should exit thereby freeing up system resources. The Job Controller will start a new channel program running when there are more messages to process. In the latter case of multiple parallel calls, there is again no need to do so. If there is an advantage to running more threads than a single call generates, then the channel's **threaddepth** channel option setting should be increased so that a single call does generate more threads. The only exception to either of these cases might be if the multiple calls are each for a different channel. Even then, however, the advantage of so doing is dubious as the same effect can be achieved through the use of multiple processes, one for each channel.

Message Processing

When **mtaDequeueStart()** is called, a communication path with the MTA Job Controller is established. The Job Controller is then asked if there are messages to be processed for the channel. Typically there will be messages to process since it is the Job Controller that normally starts channel programs, and it does so when there are queued messages in need of processing. Based upon information obtained from the Job Controller, **mtaDequeueStart()** will then begin to create non-joinable processing threads. Each processing thread immediately begins processing the queued messages.

Message Processing Procedure

To process queued messages, a processing thread takes the following steps:

1. The thread sets **ctx2** to have the value NULL:**ctx2 = NULL;** For information on the **process_message** arguments, see "The process_message() Routine"
2. The thread communicates with the Job Controller to obtain a message file to process. If there are no more message files to process, then go to "Message Processing Procedure".
3. For the message file, the thread creates a dequeue context that maintains the dequeue processing state for that message file.
4. The thread then invokes the caller-supplied **process_message** routine, passing to it the dequeue context created in "Message Processing Procedure", for example: **istat = process_message(&ctx2, ctx1, &dq_ctx, env_from, env_from_len);** For a description of the **process_message** routine, see "The process_message() Routine"
5. The **process_message** routine then attempts to process the message, ultimately removing it from the channel's queues or leaving the message file for a later processing attempt.
6. If **mtaDequeueMessageFinish()** was not called before the **process_message** routine returned, then the queued message is deferred. That is, its underlying

message file is left in the channel's queue and a later processing attempt is scheduled.

7. The dequeue context is destroyed.
8. If the **process_message** routine did not return the **MTA_ABORT** status code, then repeat this cycle starting at "Message Processing Procedure".
9. The caller-supplied **process_done** routine is called, for example:**process_done(&ctx2, ctx1);** For a description of the **process_done** routine, see "The process_done() Routine"
10. The thread exits.

process_message() Routine

This caller-supplied routine is invoked by the processing threads to do the actual processing of the messages.

The following code fragment shows the required syntax for a **process_message** routine.

```
int process_message(void      **ctx2,
                    void       *ctx1,
                    mta_dq_t   *dq_ctx,
                    const char *env_from,
                    int         env_from_len);
```

Table 6–71 lists the required arguments for a **process_message** routine, and gives a description of each.

Table 6–71 Process_Message Routine Arguments

Arguments	Description
ctx2	A writable pointer that the **process_message** routine can use to store a pointer to a per-thread context. See the description that follows for further details.
ctx1	The caller-supplied private context passed as **ctx1** to **mtaDequeueStart()**.
dq_ctx	A dequeue context created by **mtaDequeueStart()** and representing the message to be processed by this invocation of the **process_message** routine.
env_from	A pointer to the envelope **From:** address for the message to be processed. Since Internet messages are allowed to have zero length envelope **From:** addresses, this address can have zero length. The address will be NULL terminated.
env_from_len	The length in bytes of the envelope **From:** string. This length does not include any NULL terminator.

When a processing thread first begins running, it sets the value referenced by **ctx2** to NULL. This assignment is made only once per thread and is done before the first call to the **process_message** routine. Consequently, on the first call to the **process_message** routine, the following test is true:

***ctx2 == NULL**

That test will remain true until such time that the **process_message** routine itself changes the value by making an assignment to ***ctx2**. As demonstrated in the following code fragment, if the **process_message** routine needs to maintain state

across calls to itself by the same processing thread, it can allocate memory for a structure to store that state in, and then save a pointer to that memory with **ctx2**.

```
int process_message(void **ctx2, void *ctx1,
                    const char *env_from, size_t env_from_len)
{
    struct our_state_t *state;

    state = (our_state_t *)(*ctx2);
    if (!state)
    {
        /*
         * First call for this thread.
         * Allocate a structure in which to store the state
         * information
         */
        state = (our_state_t *)calloc(1, sizeof(our_state_t));
        if (!state) return(MTA_ABORT);
        *ctx2 = (void *)state;

        /*
         * Set any appropriate initial values for the state
         * structure
         */
       ...
    }
...
```

For a sample **process_message** routine, see "Example 5-2 Decoding MIME Messages Complex Example"

process_done() Routine

To assist in cleaning up state information for a thread, callers can provide a routine pointed to by the **process_done** argument.

The following code fragment shows the required syntax for a **process_done()** routine.

```
void process_done(void *ctx2,
                  void *ctx1);
```

Table 6–72 lists the arguments required for a **process_done** routine, and gives a description of each.

Table 6–72 Process_Done Routine Arguments

Required Arguments	Description
ctx2	The value of the last pointer stored by **process_message** in the **ctx2** call argument for this thread.
ctx1	The caller-supplied private context passed as **ctx1** to **mtaDequeueStart()**.

The following code fragment demonstrates the type of actions taken by a **process_done** routine.

```
void process_done(ctx2, ctx1)
{
    struct our_state_t *state = (our_state_t *)ctx2;
    if (!state)
         return;
    /*
```

```
     * Take steps to undo the state
     * (for example, close any sockets or files)
     */
    ...

    /*
     * Free the memory allocated by process_message()
     * to store the state
     */
    free(state)
}
```

Thread Creation Loop

While the processing threads are running, the thread that invoked **mtaDequeueStart()** executes a loop containing a brief pause (that is, a sleep request). Each time the **mtaDequeueStart()** thread awakens, it communicates with the Job Controller to see if it should create more processing threads. In addition, the Job Controller itself has logic to determine if more threads are needed in the currently running channel program, or if it should create additional processes to run the same channel program.

To demonstrate, the following code fragment shows pseudo code of the **mtaDequeueStart()** loop.

```
threads_running = 0
threads_max = MTA_THREAD_MAX_THREADS
attemtps    = MTA_JBC_MAX_ATTEMPTS

LOOP:
    while (threads_running &lt; threads_max)
    {

      Go to DONE if a shut down has been requested

      pending_messages = Ask the Job Controller how many
                         messsages there are to be processed

      // If there are no pending messages
      // then consider what to do next
      if (pending_messages = 0)
      {
          // Continue to wait?
          if (attempts &lt;= 0)
              go to DONE

          // Decrement attempts and wait
          attempts = attempts - 1;
          go to SLEEP
      }
      // Reset the attempts counter
      attempts = MTA_JBC_MAX_ATTEMPTS

      threads_needed = Ask the Job Controller how many
                       processing threads are needed

      // Cannot run more then threads_max threads per process
      if (threads_needed \> threads_max)
          threads_needed = threads_max

      // Create additional threads if needed
      if (threads_needed \> threads_running)
```

```
      {
         Create (threads_needed - threads_running) more threads
         threads_running = threads_needed
      }
    }

SLEEP:
    Sleep for MTA_JBC_RETRY_INTERVAL seconds
     -- a shut down request will cancel the sleep
    go to LOOP

DONE:
    Wait up to MTA_THREAD_WAIT_TIMEOUT seconds
    for all processing threads to exit

    Return to the caller of mtaDequeueStart()
```

mtaDequeueThreadId()

Return the thread ID associated with the specified dequeue context.

Syntax

```
int mtaDequeueThreadId(mta_dq_t *dq_ctx);
```

Arguments

Table 6–73 lists the **mtaDequeueThreadId()** arguments.

Table 6–73 mtaDequeueThreadId() Arguments

Arguments	Description
dq_ctx	A dequeue context created by **mtaDequeueStart()**.

Description

Each processing thread is assigned a unique integer identifier referred to as a thread ID. This thread ID is intended as a diagnostic aid when debugging channel programs. Showing it with diagnostic messages helps to differentiate the work of one thread from another in the channel's debug log file.

The thread ID can also be obtained with **mtaDequeueInfo()**.

Return Values

In the event of an error, the value **-1** is returned and **mta_errno** is set to indicate the error status code.

Table 6–74 mtaDequeueThreadId() Return Values

Error Status Code	Description
MTA_BADARGS	A NULL value was supplied for the **dq_ctx** call argument, or an invalid dequeue context was supplied for **dq_ctx**.

Example

```
mtaLog("%08d: process_message() called with dq_ctx=%p",
       mtaDequeueThreadId(dq_ctx), dq_ctx);
```

mtaDone()

Release resources used by the MTA SDK.

Syntax

```
void mtaDone(void);
```

Arguments

None

Description

Once use of the MTA SDK has been finished, **mtaDone()** should be called to release any resources used by the MTA SDK. The routine should be called while the calling process is single threaded.

Return Values

None

Example

```
mtaDone();
```

mtaEnqueueCopyMessage()

Copy a queued message to a new message being enqueued.

Syntax

```
int mtaEnqueueCopyMessage(mta_nq_t *nq_ctx,
                          mta_dq_t *dq_ctx,
                          int       rewind);
```

Arguments

Table 6–75 lists the **mtaEnqueueCopyMessage()** arguments.

Table 6–75 mtaEnqueueCopyMessage() Arguments

Arguments	Description
nq_ctx	Message submission to copy the message data to. **nq_ctx** must be an enqueue context created by **mtaEnqueueStart()**.
dq_ctx	Queued message to copy the message data from. Must be a a dequeue context created by **mtaDequeueStart()**.
rewind	Supply a value of **1** to move the read point in the queued message file to the start of the message before commencing the copy operation. Supply a value of zero to leave the message read point unchanged before copying.

Description

Intermediate processing channels often need to copy verbatim a message from a channel queue to a new message being enqueued. That is, intermediate processing

channels often re-enqueue an existing, queued message. This verbatim copy can be accomplished with **mtaEnqueueCopyMessage()**. Using this routine is significantly faster than using **mtaDequeueLineNext()** and **mtaEnqueueWriteLine()** in a read and write loop.

When **mtaEnqueueCopyMessage()** is called, the copy begins at the current read point of the queued message file associated with the supplied dequeue context, **dq_ctx**. The message file from that point to its end is copied to the new message being enqueued. To start at the beginning of the queued message (that is, to start at the beginning of its outermost header), specify a value of **1** for the **rewind** call argument. So doing is equivalent to first calling **mtaDequeueRewind()** before **mtaEnqueueCopyMessage()**.

Return Values

Table 6–76 lists the **mtaEnqueueCopyMessage()** return values.

Table 6–76 mtaEnqueueCopyMessage() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of the following reasons:# A NULL value was supplied for either the **nq_ctx** or **dq_ctx** call arguments. # Invalid contexts were passed for either or both of those call arguments. |
MTA_FCREATE	Unable to create a temporary file to hold data for the new message being enqueued.
MTA_FIO	An I/O error occurred while attempting to write data to a message file.
MTA_ORDER	Call made out of order. Either no recipients have yet been specified for the new message with **mtaEnqueueTo()**, or the recipient list of the queued message has not been completely read with **mtaDequeueRecipientNext()**.
MTA_THREAD	Simultaneous use of either the enqueue or dequeue context by two different threads was detected.

Example

The following code fragment specifies starting at the beginning of the queued message by using the **rewind** call argument.

```
mtaEnqueueMessageCopy(nq_ctx, dq_ctx, 1);
```

The code fragment that follows illustrates a second, less efficient way of copying the message.

```
mtaDequeueRewind(dq_ctx)
while (!mtaDequeueLineNext(dq_ctx, &amp;line, &amp;len))
   mtaEnqueueWriteLine(nq_ctx, line, len, NULL);
```

mtaEnqueueError()

Retrieve an extended error message.

Syntax

```
const char *mtaEnqueueError(mta_nq_t    *nq_ctx, const char **message,
                            size_t      *message_len,
```

```
                                    int          item_code);
```

Arguments

Table 6–77 lists the **mtaEnqueueError()** arguments.

Table 6–77 mtaEnqueueError() Arguments

Arguments	Description
nq_ctx	An enqueue context created by **mtaEnqueueStart()**.
message	Optional address of a pointer to receive the address of the NULL terminated error message text. A NULL value may be supplied for this argument.
message_len	Optional address of a **size_t** to receive the length in bytes of the error message text. A NULL value may be supplied for this argument.
item_code	Reserved for future use. A value of zero must be supplied for this call argument.

Description

When **mtaEnqueueTo()** returns an **MTA_NO** error message, there is often extended error information available, which takes the form of a text string suitable for writing as diagnostic output. To retrieve this information, issue **mtaEnqueueError()** immediately after receiving an **MTA_NO** error return from **mtaEnqueueTo()**.

Return Values

In the event of an error from **mtaEnqueueError()**, a **NULL** value will be returned and **mta_errno** is set to indicate the error status code. Table 6–78 lists the error status codes, and gives a description of them.

Table 6–78 mtaEnqueueError() Error Status Codes

Error Status Codes	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of the following reasons:# A NULL value was supplied for the **nq_ctx** call argument. # An invalid context was passed for **nq_ctx**. |
MTA_THREAD	Simultaneous use of the enqueue context by two different threads was detected.

Example

None

mtaEnqueueFinish()

Complete or cancel a message enqueue operation.

Syntax

```
int mtaEnqueueFinish(mta_nq_t *nq_ctx,
                     int       item_code, ...);
```

Arguments

Table 6–79 lists the **mtaEnqueueFinish()** arguments.

Table 6–79 mtaEnqueueFinish() Arguments

Arguments	Description
nq_ctx	An enqueue context created by **mtaEnqueueStart()**.
item_code	An optional list of item codes. See the description section that follows for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

Call **mtaEnqueueFinish()** to complete an enqueue operation, submitting a new message to the MTA for transport and delivery. Alternatively, call **mtaEnqueueFinish()** with the **MTA_ABORT** item code to cancel an enqueue operation without submitting a new message. In either case, when **mtaEnqueueFinish()** is called the enqueue context passed to it, **nq_ctx**, is disposed of and may no longer be used regardless of whether a success or error status code is returned.

When completing an enqueue operation, the MTA does much of the actual enqueue work, such as, performing any configured header rewriting, content transformation, and actually writing the message copy or copies to the MTA channel queues. Consequently, errors returned by this routine are typically caused by either site imposed limits (that is, the message size exceeds a site configured limit), or file system related problems (for example, the disk is full, write errors to the disk).

When **mtaEnqueueFinish()** returns an **MTA_NO** error message, there is often extended error information available. This information may be retrieved with the **MTA_REASON** item code. This extended error information takes the form of a text string suitable for writing as diagnostic output.

Before calling **mtaEnqueueFinish()** to complete an enqueue operation, be sure that the envelope recipient list has been specified with **mtaEnqueueTo()** and any header lines and content have been written with **mtaEnqueueWrite()** or **mtaEnqueueWriteLine()**.

When cancelling an enqueue operation, no message is submitted to the MTA, and any temporary files that may have been created are disposed of. To cancel an enqueue operation, specify the **MTA_ABORT** item code.

Table 6–80 lists the item codes for this routine, their additional arguments, and gives a description of each.

Table 6–80 mtaEnqueueFinish() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ ABORT	None	Cancel the current enqueue operation. The message represented by the enqueue context will not be enqueued to the MTA.
MTA_ITEM_ LIST	**mta_item_list_t *item_list**	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of zero. For further information on item list usage, see "Item Codes and Item Lists".

Table 6–80 (Cont.) mtaEnqueueFinish() Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_ REASON	**const char **errmsgsize_t *errmsg_len**	Provide the address of a string pointer to receive any extended error message information. In the event of an error associated with submitting the message to the MTA, then the MTA may return additional information. By providing this pointer, that additional information may be obtained for diagnostic purposes.This item code should be followed by two additional item codes:# The address of a pointer to receive the address of the NULL terminated error text. # The address of a **size_t** to receive the length of that error text. A value of NULL may be passed for the **errmsg_len** argument. |

Return Values

Table 6–81 lists the **mtaEnqueueFinish()** return values.

Table 6–81 mtaEnqueueFinish() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of the following reasons:# A NULL value was supplied for the **nq_ctx** call argument. # An invalid enqueue context was supplied for **nq_ctx**. # A required argument to an item code was NULL. |
MTA_FCREATE	Insufficient disk space or other I/O error encountered while attempting to create or close a message file or a temporary file.
MTA_FIO	An I/O error occurred while writing message files to the MTA channel queues or while reading from a temporary file.
MTA_NO	Error terminating the message temporary file, there appears to be insufficient disk space to write the message copies, or there is a problem with a configured content scanner (for example, a virus or spam filter).
MTA_NOSUCHITEM	An invalid item code was supplied.
MTA_ORDER	The call was made out of order. Either no envelope recipient addresses have been specified or no message content has been provided.
MTA_THREAD	Simultaneous use of the enqueue context by two different threads was detected.

Note: In case of an error, the **MTA_REASON** item code can be used to receive extended error message information

As shown in the preceding table, in the case of an error, the **MTA_REASON** item code can be used to receive extended error message information

Example

See "A Simple Example of Enqueuing a Message".

mtaEnqueueInfo()

Obtain information associated with an ongoing message enqueue.

Syntax

```
int mtaEnqueueInfo(mta_nq_t *nq_ctx,
                   int       item_code, ...);

            int mtaEnqueueInfo(mta_nq_t *nq_ctx,
```

Arguments

Table 6–82 lists the **mtaEnqueuInfo()** arguments.

Table 6–82 mtaEnqueueInfo() Arguments

Arguments	Description
nq_ctx	An enqueue context created by **mtaEnqueueStart()**.
item_code	An optional list of item codes. See the description section that follows for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

Information associated with an ongoing message enqueue operation may be obtained with **mtaEnqueueInfo()**. The information to obtain is specified through the use of item codes. Arguments to the item codes provide memory addresses through which to return the requested data.

String pointers returned by **mtaEnqueueInfo()** are only valid during the life of the enqueue context. Once the enqueue has been completed, the associated pointers are no longer valid.

Table 6–83 lists the item codes for this routine, their additional arguments, and gives a description of each.

Table 6–83 mtaEnqueueInfo() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ ALIAS_ EXPAND	**size_t *value**	Return the setting of the alias expansion flag. Normally, this flag has a nonzero value that indicates that alias expansion should be done for all envelope recipient addresses. When the flag has a value of zero, alias expansion will not be performed. The value of the flag is set with the **mtaEnqueueStart()** routine.This item code must be followed by one additional argument: the address of **size_t** to store the setting's value in.
MTA_ADR_ SORT	**size_t *value**	Obtain the setting of the address sorting flag. Normally, this flag has a non-zero value that indicates that the list of envelope recipients written to each message copy in the MTA channel queues are to be sorted in ascending order based upon US-ASCII ordinal values. When this flag has a value of zero, the list of envelope recipient addresses will not be sorted. This item code must be followed by one additional argument: the address of **size_t** to store the setting's value in.

Table 6–83 (Cont.) mtaEnqueueInfo() Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_ CHANNEL	**char **channelsize_t *channel_len**	Obtain the name of the channel that this message is being enqueued by.This item code must be followed by two additional call arguments:# The address of a pointer to receive the address of the NULL terminated channel name. # The address of a **size_t** to receive the length of the channel name. A NULL value may be passed for the **channel_len** argument. |
MTA_ DELIVERY_ FLAGS	**size_t *dflags**	Return the envelope delivery flags set for the entire message by **mtaEnqueueStart()**.This item code must be followed by one additional call argument: the address of a **size_t** to receive the delivery flag setting.
MTA_ENV_ FROM	**const char **env_fromsize_ t *env_from_len**	Retrieve the envelope **From:** address specified when the enqueue was started with **mtaEnqueueStart()**.This item code must be followed by two additional call arguments:# The address of a pointer to receive the address of the NULL terminated envelope **From:** address. # The address of a **size_t** to receive the length of that address. A NULL value may be passed for the **env_from_len** argument. |
MTA_ENV_ ID	**const char **env_idsize_t *env_id_len**	Obtain the envelope ID specified with **mtaEnqueueStart()**.This item code must be followed by two additional call arguments:# The address of a pointer to receive the address of the NULL terminated envelope ID. # The address of a **size_t** to receive the length of that envelope ID. A NULL value may be passed for the **env_ id_len** argument. |
MTA_ EXPAND_ LIMIT	**size_t *value**	Retrieve the expand limit setting specified with **mtaEnqueueStart()**. The returned value will be a positive integer value. When no expand limit has been set, the returned value will be a large integer value (for example, 2,147,483,647 on 32-bit processors).This item code must be followed by one additional argument: the address of a **size_t** to store the setting's value in.
MTA_ FRAGMENT _BLOCKS	**size_t *value**	Obtain the value, if any, specified for the **MTA_ FRAGMENT_BLOCKS** setting when the message enqueue was initiated. The returned value will be a positive integer value. When no value was set, the returned value will be a large integer value (for example, 2,147,483,647 on 32-bit processors).This item code must be followed by one additional argument: the address of a **size_t** to store the setting's value in.
MTA_ FRAGMENT _LINES	**size_t *value**	Obtain the value specified for the **MTA_FRAGMENT_ LINES** setting when the message enqueue was initiated. The returned value will be a positive integer value. When no value was set, the returned value will be a large integer value (for example, 2,147,483,647 on 32-bit processors).This item code must be followed by one additional argument: the address of a **size_t** to store the setting's value in.

Table 6–83 (Cont.) mtaEnqueueInfo() Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_ NOTIFY_ FLAGS MTA_ NOTIFY_ FAILURE MTA_ NOTIFY_ CONTENT_ FULL This item code must be followed by one additional call argument: the address of a **size_t** to receive the setting of the delivery status notification flags.	**size_t *nflags**	Return the delivery status notification flags set for the entire message when the enqueue was started. The returned value is a bit map constructed using the **MTA_ NOTIFY_** constants defined in **mtasdk.h**. If no setting was effected with **mtaEnqueueStart()**, then the returned value will be the MTA default of: **MTA_NOTIFY_DELAY** {{

Return Values

Table 6–84 lists the **mtaEnqueueInfo()** return values.

Table 6–84 mtaEnqueueInfo() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of the following reasons:# A NULL value was supplied for the **nq_ctx** call argument. # An invalid enqueue context was supplied for **nq_ctx**. # A required argument to an item code was NULL. |
MTA_NOSUCHITEM	An invalid item code was specified.
MTA_THREAD	Simultaneous use of the enqueue context by two different threads was detected.

Example

The following code fragment obtains the name of the channel used as the source channel for the enqueue.

```
mta_nq_t *nq;
const char *channel;

mtaEnqueueStart(&amp;nq, "sue@siroe.com", 0, 0);
mtaEnqueueInfo(nq, MTA_CHANNEL, &amp;channel, NULL, 0);
printf("Source channel = %s\n", channel);
```

mtaEnqueueStart()

Initiate a message submission.

Syntax

```
int mtaEnqueueStart(mta_nq_t   **nq,
                    const char *env_from,
                    size_t      env_from_len,
                    int         item_code, ...);
```

Arguments

Table 6–85 lists the **mtaEnqueueStart()** arguments.

Table 6–85 mtaEnqueueStart() Arguments

Arguments	Description
nq_ctx	On a successful return, a pointer to an enqueue context created by **mtaEnqueueStart()**. This enqueue context represents the message enqueue operation initiated by the call.
env_from	Optional pointer to the address to use as the envelope **From:** address for the message being submitted. The address must be compliant with RFC 2822. When used as an envelope address, the MTA will reduce it to an RFC 2821 compliant transport address. The string must be NULL terminated if a value of zero is passed for **env_from_len**. The length of this string, not including any NULL terminator, may not exceed **ALFA_SIZE** bytes.A value of NULL may be supplied for this argument. When that is done, the **env_from_len** argument is ignored and an empty envelope **From:** address is used for the message submission.
env_from_len	The length in bytes, not including any NULL terminator, of the envelope **From:** address supplied with **env_from**. If a value of zero is passed for this argument, then the envelope **From:** address string must be NULL terminated.
item_code	An optional list of item codes. See the description section that follows for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

To submit a message to the MTA for delivery, an enqueue operation must be initiated. This is achieved by calling **mtaEnqueueStart()**. When the call is successful, an enqueue context representing the enqueue operation will be created and a pointer to the context returned via the **nq_ctx** call argument. This context must then be used to specify the message's envelope recipient list and content, both header and body. Once the recipient list and content have been specified, the submission may be completed with **mtaEnqueueFinish()**. That same routine is also used to cancel an enqueue operation. For further information on message enqueue processing, see "Basic Steps to Enqueue Messages".

Enqueue contexts are disposed of with **mtaEnqueueFinish()**, either as part of completing or cancelling a message enqueue operation.

When initiating an enqueue operation, the envelope **From:** address for the message should be specified with the **env_from** and **env_from_len** call arguments, or through use of a dequeue context with the **MTA_DQ_CONTEXT** item code. In either case, it is important to keep in mind the usage of the envelope **From:** address. MTAs transporting the message use it as a return path, that is, the address to which

notifications about the message should be returned. Specifically, it is the address to which the message will be returned in the form of a non-delivery notification (NDN) should the message prove undeliverable. It is also the address to which any delivery status notifications (DSNs) will be sent. As such, the envelope **From:** address specified should be an address suitable for receiving such notifications.

> **Note:** Automatically generated messages such as NDNs and DSNs are required to have an empty envelope **From:** address, that is, a zero length address. These rules are mandated by Internet standards so as to prevent broad classes of looping messages. It is imperative that they be observed; failure to do so may result in exponentially growing mail loops that affect not only your own mail system but possibly mail systems of other sites with which you exchange mail.

When explicitly specifying the envelope **From:** address via the **env_from** and **env_from_len** call arguments, note the following points:

- The length of the address may not exceed 256 bytes. This is the length limit imposed by RFCs 2821 and 2822. It is also the size denoted by the **ALFA_SIZE** constant.
- Older MTAs may not support envelope addresses of lengths exceeding 129 bytes. This is the length limit imposed by RFC 821.
- To specify an empty envelope **From:** address, supply an empty string for **env_from** and a length of zero for **env_from_len**, or supply a value of NULL for **env_from** and any value for **env_from_len**.

When using a dequeue context to supply the envelope **From:** address, simply supply a value of NULL and zero for, respectively, the **env_from** and **env_from_len** call arguments. Be sure to also supply the dequeue context with the **MTA_DQ_CONTEXT** item code. For example:

```
ires = mtaEnqueueStart(&nq, NULL, 0, MTA_DQ_CONTEXT, dq, 0);
```

If the submitted message lacks a **From:** header line, then the address supplied as the envelope **From:** address will also be used to generate a **From:** header line. This is the reason why **mtaEnqueueStart()** allows an RFC 2822 compliant address to be supplied for the envelope **From:** address. When placing the supplied address into the envelope, the MTA reduces it to an RFC 2821 compliant address (for example, removes any RFC 2822 phrases or comment fields).

When submitting a message, the MTA requires a source channel to associate with the enqueue operation. By default, the name of the source channel will be derived from the **PMDF_CHANNEL** environment variable. However, this may be overridden one of two ways: by supplying a dequeue context with the **MTA_DQ_CONTEXT** item code, or by explicitly specifying the channel name with the **MTA_CHANNEL** item code. Use of a dequeue context implicitly specifies the source channel name to be the name of the channel associated with the dequeue context.

> **Note:** An explicitly specified channel name will take precedence over a channel name specified with a dequeue context.

As part of initiating a message submission, item codes may be used to specify additional envelope information for the message as well as select non-default values for MTA parameters that influence message enqueue processing.

Table 6–86 lists the items codes for this routine, their additional arguments, and gives a description of each.

Table 6–86 mtaEnqueueStart() Item Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ ALIAS_ EXPAND	None	When this item code is specified, each envelope recipient address is allowed to undergo alias expansion (for example, mailing list expansion). This is the default behavior.
MTA_ ALIAS_ NOEXPAN D	None	Use of this item code inhibits alias expansion for the envelope recipient addresses. The default behavior is to permit alias expansion.
MTA_ADR_ NOSORT	None	Inhibit sorting of the envelope recipient list in the message copies written to the MTA channel queues. By default, the envelope recipient address list is sorted. Use this option if it is imperative that the envelope recipients be processed in some specific order. Maintaining the order requires control of all MTA channels that the message will pass through.
MTA_ADR_ SORT	None	Allow the envelope recipient list to be sorted in the message copies written to the MTA channel queues. This is the default behavior.
MTA_ CHANNEL	**char *channelsize_t channel_len**	Explicitly specify the name of the channel under which to enqueue this message. That is, explicitly specify the name of the source channel to use for this message submission. The name specified will override any name implicitly specified with the **MTA_DQ_CONTEXT** item code.This item code must be followed by two additional call arguments:# The address of the string containing the channel name. # The length in bytes of that channel name. If a value of zero is specified for the length, then the channel name string must be NULL terminated. |
MTA_ DELIVERY_ FLAGS	**size_t dflags**	Specify additional envelope delivery flags to set for this message. The logical OR of any existing setting and the value here supplied will be used for the message's delivery flag setting. In general, the delivery flag setting associated with a message will be the logical OR of the values set by each channel a message has travelled through. Note that channels also can set this value with the **deliveryflags** channel option. When this item code is not used, the delivery flags inherited from a supplied dequeue context will be used. If no dequeue context is supplied, then the value of the delivery flags will be set to zero.This item code should be followed by an additional call argument: the value to combine with any existing setting.
MTA_ DELIVERY_ FLAGS_ ABS	**size_t dflags**	Ignore any previous envelope delivery flag setting for the message and replace the setting with the value specified with this item code.This item code should be followed by an additional call argument: the delivery flag setting to effect.

Table 6–86 (Cont.) mtaEnqueueStart() Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_DQ_ CONTEXT	**mta_dq_t *dq_ctx**	When a dequeue context is supplied with this item code, the message submission will take all of its envelope fields, except for the recipient list, from the envelope of the queued message represented by the dequeue context, including the envelope **From:** field. These assumed settings can then be overridden on an individual basis through the use of other item codes, and the **env_from** and **env_from_len** call arguments.Use of this item code changes the defaults for the envelope fields from the MTA defaults to the values used in the dequeue context.Intermediate processing channels are strongly encouraged to use this item code. Use of this feature allows envelope information to be automatically copied from the queued message being processed to the new message that will be enqueued as a result.This item code must be followed by one additional argument: the pointer to the dequeue context to use.
MTA_ENV_ ID	**const char *env_idsize_ t env_id_len**	Explicitly specify an envelope ID string for the message. The supplied value must conform to the syntax of an **xtext** object in RFC 1891 and may not have a length exceeding 100 bytes. The value specified with this item code will override any value implicitly specified with the **MTA_DQ_CONTEXT** item code. If no value is supplied either explicitly or implicitly, then the MTA will generate a unique envelope ID for the message.This item code must be followed by two additional call arguments:# The address of the envelope ID string. # The length in bytes of that string. If a value of zero is supplied for the length, then the string must be NULL terminated. |
MTA_ EXPAND_ LIMIT	**size_t limit**	If the message has more envelope recipients than the specified limit, then processing of the recipient list (that is, alias expansion) will be deferred. This deferral is performed by enqueuing the message to the reprocess channel. At a later time, and running in a separate process, the reprocess channel will complete the processing of the envelope recipient list.This item code must be followed by one additional argument: the limit to impose. By default, no limit is imposed.

Table 6–86 (Cont.) mtaEnqueueStart() Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_ FRAGMEN T_BLOCKS	**size_t blocks**	A large enqueued message may automatically be fragmented into several, smaller messages using MIME's **message/partial** content type. At the destination MTA system, these smaller messages may automatically be re-assembled back into one single message. The **MTA_FRAGMENT_BLOCKS** item code allows specification of a size threshold for which messages larger than the threshold will automatically be fragmented. The limit specified is measured in units of blocks. (By default, a block is 1024 bytes.) However, sites may change that size with the MTA **BLOCK_SIZE** option. Consequently, code using this option should use the **mtaBlockSize()** option should they need to convert some other unit to blocks.This item code must be followed by one additional argument: the block size threshold to impose. By default, no threshold is imposed.
MTA_ FRAGMEN T_LINES	**size_t lines**	A large enqueued message can be automatically fragmented into several, smaller messages using the MIME content type **message/partial**. At the destination MTA system, these smaller messages can be automatically re-assembled back into one single message. The **MTA_FRAGMENT_LINES** item code allows specification of a line count threshold for which messages exceeding the threshold will automatically be fragmented.This item code must be followed by one additional argument: the line count threshold to impose. By default, no threshold is imposed.

Table 6–86 (Cont.) mtaEnqueueStart() Item Codes, Additional Arguments, and

Item Codes	Additional Arguments	Description
MTA_ NOTIFY_ FLAGS MTA_ NOTIFY_ FAILURE MTA_ NOTIFY_ CONTENT_ FULL Flags for individual recipient address may be specified when **mtaEnqueue To()** is called.This item code must be followed by one additional call argument: the address of an integer to receive the setting of the delivery status notification flags.	**size_t nflags**	Specify the delivery status notification flags to be set for the entire message. The specified value is a bit map constructed using the **MTA_NOTIFY_** constants defined in **mtasdk.h**. If no setting is made, then the value from a supplied dequeue context will be used. If no dequeue context is supplied, then the MTA default value is used. The default value is: MTA_NOTIFY_DELAY

Return Values

Table 6–87 lists the **mtaEnqueueStart()** return values.

Table 6–87 mtaEnqueueStart() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of the following reasons:# A NULL value was supplied for the **nq_ctx** call argument. # An invalid enqueue context was supplied for **nq_ctx**. # A required argument to an item code was NULL. ǀ
MTA_NO	Unable to determine the channel name from the **PMDF_ CHANNEL** environment variable,
MTA_NOMEM	Insufficient virtual memory.
MTA_NOSUCHCHAN	Specified channel name does not exist in the MTA configuration.
MTA_NOSUCHITEM	An invalid item code was specified.
MTA_STRTRUERR	The supplied envelope From: address is too long; it may not exceed a length of **ALFA_SIZE** bytes. Or the supplied channel name has a length exceeding **CHANLENGTH** bytes.

Example

This routine is used as part of "Example 5-2 Decoding MIME Messages Complex Example".

mtaEnqueueTo()

Add an envelope recipient to a message being submitted.

Syntax

```
int mtaEnqueueTo(mta_nq_t   *nq_ctx,
                 const char *to_adr,
                 size_t      to_adr_len,
                 int         item_code, ...);
```

Arguments

Table 6–88 lists the **mtaEnqueueTo()** arguments.

Table 6–88 mtaEnqueueTo() Arguments

Arguments	Description
nq_ctx	Pointer to an enqueue context created with **mtaEnqueueStart()**.
to_adr	An address to add to the message being enqueued. The address must be compliant with RFC 2822. When used as an envelope address, the MTA will reduce it to an RFC 2821 compliant transport address. If a value of zero is passed for **to_adr_len** the address string must be NULL terminated. The length of this string, not including any NULL terminator, may not exceed **ALFA_SIZE** bytes.
to_adr_len	The length in bytes, not including any NULL terminator, of the address supplied with **to_adr**. If a value of zero is passed for this argument, then the address string must be NULL terminated.
item_code	An optional list of item codes. See the description section below for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

After initiating a message enqueue operation with **mtaEnqueueStart()**, the envelope recipient list for the message must to be constructed. This list is the actual list of recipients to which the message is to be delivered. A message must have at least one envelope recipient address; otherwise, there is no one to deliver the message to. In the envelope there is no distinction between **To:**, **Cc:**, or **Bcc:** addressees. Additionally, the list of addressees appearing in the message's header need not be the same as those appearing in the envelope. This is the case with list-oriented mail. The address in the message's header is often the list's mail address; whereas, the addresses in the envelope are the those of the list's individual members.

By default, when an address is added to a message with **mtaEnqueueTo()**, it is added as both an envelope recipient address as well as a **To:** addressee in the message's **To:** header line. The address is therefore considered to be an active transport address as well as a header address. This case corresponds to the **MTA_TO** item code. To instead mark an active transport address for addition to either a **Cc:** or **Bcc:** header line, use the **MTA_CC** or **MTA_BCC** item code.

Addresses that only appear in the message's header are sometimes referred to as inactive addresses. Such addresses added with **mtaEnqueueTo()** may be noted as such with the **MTA_HDR_TO**, **MTA_HDR_CC**, and **MTA_HDR_BCC** item codes. They can also be manually added by constructing the **To:**, **Cc:**, or **Bcc:** header lines with **mtaEnqueueWrite()** or **mtaEnqueueWriteLine()**.

> **Note:** The MTA SDK will automatically generate multiple message copies when **Bcc:** recipients exist for the message. Specifically, when a message has N envelope recipient addresses which are **Bcc:** recipients, the MTA SDK will automatically generate N+1 message copies: one copy for each of the **Bcc:** recipients and an additional copy for the remaining, non-**Bcc:** recipients. Each copy for a **Bcc:** recipient will only disclose that **Bcc:** recipient in the message's header. The message copy for all of the non-**Bcc:** recipients will disclose none of the **Bcc:** recipients in its header

An address may be added as only an active transport address without addition to any header line. This is done with the **MTA_ENV_TO** item code. This item code should be used by intermediate processing channels that copy verbatim the outer message header from the old message to the new, which prevents duplication of addresses in the new message's header.

When an active transport address is added to a message, it is possible that the MTA will reject the address. For example, the address can be rejected when there is a mapping table, such as the **SEND_ACCESS** mapping table. When an address is rejected by the MTA, extended error text is made available by the MTA. This extended information can be captured through use of the **MTA_REASON** item code.

Table 6–89 lists the item codes for this routine, their additional arguments, and gives a description of each.

Table 6–89 mtaEnqueueTo() tem Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_BCC	None	The address is an active transport address that should also appear in a **Bcc:** header line. The address will be added to both the envelope recipient list as well as the message's header. For further information about **Bcc:**, see the note under "Description".
MTA_CC	None	The address is an active transport address that should also appear in a **Cc:** header line. As such, the address will be added to both the envelope recipient list as well as the message's header.
MTA_ DELIVERY_ FLAGS	**size_t dflags**	Specify additional envelope delivery flags to set for this recipient. The logical OR of any existing setting for the recipient and the value here supplied will be used for the recipient's delivery flag setting. The existing setting for the recipient will be either the message's setting, which was set with **mtaEnqueueStart()**, or any setting copied over from the dequeue context for this recipient with the **MTA_ DQ_CONTEXT** item code.This item code should be followed by one additional call argument: the value to combine with any existing setting.

Table 6–89 (Cont.) mtaEnqueueTo() tem Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ DELIVERY_ FLAGS_ ABS	**size_t dflags**	Ignore any previous envelope delivery flag setting for the recipient and replace the setting with the value specified with this item code.This item code should be followed by one additional call argument: the delivery flag setting to effect.
MTA_DQ_ CONTEXT	**mta_dq_t *dq_ctx**	When a dequeue context is supplied using this item code, the specified envelope recipient address is compared to the envelope recipient list for the queued message represented by the dequeue context. If a match is found, envelope fields for the recipient are copied from the queued message to the new message being enqueued. If no match is found, an **MTA_NO** error status is returned.This item code must be followed by one additional argument: the pointer to the dequeue context to use.
MTA_ENV_ TO	None	The address is an active transport address; add it to the envelope recipient list. Do not add it to any header lines. This designation is often used by intermediate processing channels.
MTA_ HDR_BCC	None	The address is not an active transport address; do not add it to the envelope recipient list. The address should, however, be added to a **Bcc:** header line. Note that since a **Bcc:** header line is usually only placed in the message copy destined to the **Bcc:** recipient, use of this item code only arises when the **Bcc:** recipient's header address differs from their transport address and, consequently, the two need to be added with separate calls to **mtaEnqueueTo()**.
MTA_ HDR_CC	None	The address is not an active transport address; do not add it to the envelope recipient list. The address should, however, be added to a **Cc:** header line.
MTA_ HDR_TO	None	The address is not an active transport address; do not add it to the envelope recipient list. The address should, however, be added to a **To:** header line.
MTA_ NOTIFY_ FLAGS	**size_t nflags**	Delivery status notification flags specific to this envelope recipient address. A value specified with this item code overrides any setting made for the message itself when the enqueue context was created. It also overrides any value inherited from a dequeue context. Note that this item code has no effect when **MTA_HDR_BCC**, **MTA_HDR_CC**, or **MTA_HDR_TO** is specified; notification flags only apply to active transport addresses. For further details, see the description of this item code for mtaEnqueueStart().This item code must be followed by one additional call argument: the address of an integer to receive the setting of the delivery status notification flags.
MTA_ ORCPT_TO	**const char *orcptsize_t orcpt_len**	Specify the original envelope recipient address in RFC 1891 original-recipient address format (for example, **rfc822;sue@siroe.com** for **sue@siroe.com**).This item code must be followed by two additional arguments:# The pointer to the original recipient address. # The length in bytes of that address. If a value of zero is supplied for the length, then the address string must be NULL terminated. |

Table 6–89 (Cont.) mtaEnqueueTo() tem Codes, Additional Arguments, and Descriptions

Item Codes	Additional Arguments	Description
MTA_ REASON	**const char **errmsgsize_t *errmsg_len**	Provide the address of a string pointer to receive any extended error message information. In the event of an error associated with submitting the recipient to the MTA, then the MTA may return additional information. By providing this pointer, that additional information may be obtained for diagnostic purposes.This item code should be followed by two additional item codes:# The address of a pointer to receive the address of the NULL terminated error text. # The address of a **size_t** to receive the length of that error text. A value of NULL can be passed for the **errmsg_len** argument. |
MTA_TO	None	The address is an active transport address that should also appear in a **To:** header line. This is the default interpretation of addresses added with **mtaEnqueueTo()**.

Return Values

Table 6–90 lists the **mtaEnqueueTo()** return values.

Table 6–90 mtaEnqueueTo() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of the following reasons:# A NULL value was supplied for the **nq_ctx** call argument. # An invalid enqueue context was supplied for **nq_ctx**. # A required argument to an item code was NULL. |
MTA_NO	If **MTA_DQ_CONTEXT** was specified, then the supplied envelope **To:** address does not match any envelope recipient address in the queued message represented by the supplied dequeue context. Otherwise, the MTA rejected the envelope recipient address. It could be syntactically invalid, refused by a mapping table, such as **SEND_ACCESS**. Consider using the **MTA_REASON** item code.
MTA_NOSUCHITEM	An invalid item code was specified.
MTA_ORDER	The call was made out of order: the message's envelope recipient list has already been terminated by a call to **mtaEnqueueWrite()** or **mtaEnqueueWriteLine()**.
MTA_STRTRUERR	The supplied envelope **To:** address or original envelope To: address is too long. Neither may exceed a length of **ALFA_SIZE** bytes.

Example

This routine is used in "Example 5-2 Decoding MIME Messages Complex Example".

mtaEnqueueWrite()

Write message data to the message being submitted.

Syntax

```
int mtaEnqueueWrite(mta_nq_t    *nq_ctx,
                    const char  *str1,
                    size_t       len1,
                    const char  *str2, ...);
```

Zero or more string pointer-length pairs can be supplied to this routine. The list of pairs must be terminated by a NULL call argument.

Arguments

Table 6–91 lists the **mtaEnqueueWriter()** arguments.

Table 6–91 mtaEnqueueWrite() Arguments

Arguments	Description
nq_ctx	Pointer to an enqueue context created with **mtaEnqueueStart()**.
str1	Pointer to a string of text to write to the message. The string must be NULL terminated if a value of zero is passed for **len1**.
len1	The length in bytes, not including any NULL terminator, of the string **str1**. If a value of zero is passed for this argument, then the string **str1** must be NULL terminated.
str2	Pointer to a second string of text to write to the message. The string must be NULL terminated if a value of zero is passed for **len2**. If only supplying a single string, then pass a NULL value for this argument.

Description

After a message's list of envelope recipient addresses has been supplied with **mtaEnqueueTo()**, the message itself must be supplied. This is done by repeatedly calling **mtaEnqueueWrite()**. First the message's header should be supplied, followed by a blank line, followed by any message content. Each line of message data must be terminated by a US-ASCII line-feed character (**0x0A**). Each call to **mtaEnqueueWrite()** can supply one or more bytes of the message's data. Unlike **mtaEnqueueWriteLine()**, a single call to **mtaEnqueueWrite()** does not necessarily correspond to a single, complete line of message data; it could correspond to a partial line, a complete line, multiple lines, or even one or more complete lines plus a partial line. This flexibility with **mtaEnqueueWrite()** exists because it is up to the caller to supply the message line terminators. Calling either **mtaEnqueueWrite()** or **mtaEnqueueWriteLine()** terminates the message's envelope recipient list. Once either of these routines have been called, **mtaEnqueueTo()** can no longer be called for the same enqueue context.

Return Values

Table 6–92 lists the **mtaEnqueueWrite()** return values.

Table 6–92 mtaEnqueueWrite() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of the following reasons:# A NULL value was supplied for the **nq_ctx** call argument. # An invalid enqueue context was supplied for **nq_ctx**, or a required argument to an item code was NULL. |

Table 6–92 (Cont.) mtaEnqueueWrite() Return Values

Return Values	Description
MTA_FCREATE	Unable to create a disk file.
MTA_FIO	Error writing to a disk.
MTA_ORDER	Call made out of order. No envelope recipient addresses have been supplied.
MTA_THREAD	Simultaneous use of the enqueue context by two different threads was detected.

Example

The code fragment that follows shows two ways to produce the same results. They both write two header lines to the message:

```
mtaEnqueueWrite(nq, "From: sue@siroe.com\n", 0, NULL);
mtaEnqueueWrite(nq, "Subject: test\n", 0, NULL);


mtaEnqueueWrite(nq, "From: sue@siroe.com\nSubject: test\n", 0,
               NULL);
```

The following code fragment shows the two header lines output by each code fragment in the preceding code example.

```
From: sue@siroe.com
Subject: test
```

This code fragment demonstrates how to terminate the message header by writing a blank line.

```
mtaEnqueueWrite(nq, "\n", 0, NULL);
```

The following code fragment shows a single call to mtaEnqueueWrite() that writes out an entire header, including the terminating blank line.

```
mtaEnqueueWrite(nq, "Date: today\nFrom: sue@siroe.com\n"
                    "To: bob@siroe.com\nSubject: test\n\n", 0,
                    NULL);
```

The following code example shows an alternate way of writing the routine call, but with one pair per line.

```
mtaEnqueueWrite(nq, "Date: today\n", 0,
                    "From: sue@siroe.com\n", 0,
                    "To: bob@siroe.com\n", 0,
                    "Subject: test\n", 0,
                    "\n", 0,
                    NULL);
```

mtaEnqueueWriteLine()

Write a complete, single line of message data to the message being submitted.

Syntax

```
int mtaEnqueueWrite(mta_nq_t    *nq_ctx,
                    const char  *str1,
                    size_t       len1,
```

```
                                const char  *str2, ...);
```

Zero or more string pointer-length pairs can be supplied to this routine. The list of pairs must be terminated by a NULL call argument.

Arguments

Table 6–93 lists the **mtaEnqueueWriteLine()** arguments.

Table 6–93 mtaEnqueueWriteLine() Arguments

Arguments	Description
nq_ctx	Pointer to an enqueue context created with **mtaEnqueueStart()**.
str1	Pointer to a string of text to write to the message. The string must be NULL terminated if a value of zero is passed for **len1**.
len1	The length in bytes, not including any NULL terminator, of the string **str1**. If a value of zero is passed for this argument, then the string **str1** must be NULL terminated.
str2	Pointer to a second string of text to write to the message. The string must be NULL terminated if a value of zero is passed for **len2**. If only supplying a single string, then pass a NULL value for this argument.

Description

After a message's list of envelope recipient addresses has been supplied with **mtaEnqueueTo()**, the message itself must be supplied. This can be done by repeatedly calling **mtaEnqueueWriteLine()**. First the message's header should be supplied, followed by a blank line, followed by any message content. Each call to this routine must supply a single, complete line of the message. The line should not include a line-feed terminator as **mtaEnqueueWriteLine()** will supply the terminator automatically.

Calling **mtaEnqueueWriteLine()** terminates the message's envelope recipient list. Once the routine is called, **mtaEnqueueTo()** can no longer be called for the same enqueue context.

Return Values

Table 6–94 lists the **mtaEnqueueWriteLine()** return values.

Table 6–94 mtaEnqueueWriteLine() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	This value is returned for one of the following reasons:# A NULL value was supplied for the **nq_ctx** call argument. # An invalid enqueue context was supplied for **nq_ctx**, or a required argument to an item code was NULL. |
MTA_FCREATE	Unable to create a disk file.
MTA_FIO	Error writing to a disk.
MTA_ORDER	Call made out of order. No envelope recipient addresses have been supplied.
MTA_THREAD	Simultaneous use of the enqueue context by two different threads was detected.

Example

This code fragment writes out two header lines.

```
mtaEnqueueWriteLine(nq, "From: sue@siroe.com", 0, NULL);
mtaEnqueueWriteLine(nq, "Subject: test", 0, NULL);
```

This code fragment shows the header output as a result of the preceding code example.

```
From: sue@siroe.com
Subject: test
```

The following code fragment shows how to terminate the header by writing a blank line.

```
mtaEnqueueWriteLine(nq, "", 0, NULL);
```

The following code fragment that produces a **Date:** header line.

```
char buf[64];

mtaEnqueueWriteLine(nq,
                    "Date: ", 0,
                     mtaDateTime(buf, NULL, sizeof(buf), 0), 0,
                     NULL);
```

mtaErrno()

Obtain the last returned error status for the calling thread.

Syntax

```
int mtaErrno(void);
```

Arguments

None

Description

When an MTA SDK routine is called by a processing thread and returns an error status code, the SDK saves that status code in thread-specific data. The same processing thread can obtain the most recently saved status code for its own thread of execution by calling **mtaErrno()**.

For convenience purposes, the **mtasdk.h** header file also defines **mta_errno** as a macro that calls **mtaErrno()**. Specifically:

```
#define mta_errno mtaErrno()
```

Return Values

The last error return status code returned by an MTA SDK routine called by this processing thread.

For a description of the MTA SDK error status codes, see "Error Status Codes Summary".

Example

The following code fragment demonstrates how to obtain the most recent error status code for its own thread.

```
if (!mtaEnqueueStart(&amp;nq, from_adr, 0, 0))
    printf("Error returned: %d\n", mtaErrno());
```

mtaInit()

Initialize the MTA SDK.

Syntax

```
int mtaInit(int item_code, ...);
```

Arguments

Table 6–95 lists the **mtaInit()** arguments.

Table 6–95 mtaInit() Arguments

Arguments	Description
item_code	An optional list of item codes. See the description section that follows for a list of item codes. The list must be terminated with an integer argument with value **0**.

Description

Call the **mtaInit()** routine to initialize the MTA SDK. As part of the initialization process, the SDK will load the MTA configuration. This loading process will be the typical cause of initialization failures; either there's an error in a configuration file, a missing but required configuration file, or a configuration file can't be accessed for reading. To prevent that last error case, ensure that your programs run under a UID that has read access to the MTA configuration files, especially the compiled configuration file produced by the **imsimta cnbuild** utility.

While there is no benefit to doing so, it is safe to call **mtaInit()** multiple times, either before or after calling **mtaDone()**. (To de-initialize the SDK, use **mtaDone()**.)

Although the MTA SDK is self-initializing, the initialization must occur while the process is single-threaded. As such, multi-threaded programs must call **mtaInit()** and must do so while still single threaded.

When the SDK is initialized, the SDK can be told using an item code whether or not the calling program will be functioning as an interactive utility or not. When being used by an interactive utility, such as a management utility or a user agent, the SDK ensures that accounting files are closed after every operation that records accounting information. This prevents the accounting file from being left open by a single process for long periods of time. To specify that the SDK will be used by an interactive utility, specify the **MTA_INTERACTIVE** item code. By default, the SDK assumes that it will be run by a channel program or other program that wishes to achieve maximum performance while using the SDK. This corresponds to the **MTA_CHANNEL** item code. Also, when the SDK self-initializes itself, it assumes **MTA_CHANNEL** and not **MTA_INTERACTIVE**. As part of initializing the SDK, a number of diagnostic facilities can be enabled. These are enabled using the **MTA_DEBUG_** item codes described in Table 6–96. These diagnostic facilities may also be enabled at any time using the **mtaDebug()** routine.

Table 6–96 mtaInit() Item Codes, Additional Arguments, and Descriptions

Item Code	Additional Arguments	Description
MTA_ CHANNEL	None	Indicate that the SDK is being used by a channel program or other non-interactive program. By default this is the assumed usage. Interactive programs should use the **MTA_INTERACTIVE** item code.
MTA_ DEBUG_ DECODE	None	Enable diagnostic output from the low-level MIME decoding routines used by the MTA SDK. This diagnostic output may prove helpful when attempting to understand any MIME conversions that occur either when enqueuing messages to the MTA and the destination channel is configured to invoke MIME conversions (for example, marked with channel options such as **thurman** or **inner**), or when using the SDK message decoding routine, **mtaDecodeMessage.()**
MTA_ DEBUG_ DEQUEUE	None	Enable diagnostic output from the low-level queue processing routines used by the MTA SDK. Use this diagnostic output when attempting to understand issues surrounding reading and processing of queued message files. This diagnostic output will not help diagnose the selection of queued messages as that is handled by a separate process: the MTA Job Controller.Enabling this diagnostic output is equivalent to setting **DEQUEUE_DEBUG=1** in the MTA option file, **option.dat**.
MTA_ DEBUG_ ENQUEUE	None	Enable diagnostic output from the low-level message enqueue routines used by the MTA SDK. Enqueue diagnostics can be used to diagnose the address rewriting process, destination channel selection, header processing, and other types of processing that occurs when a message is enqueued to the MTA.Enabling this diagnostic output is equivalent to setting **MM_DEBUG=5** in the MTA option file.
MTA_ DEBUG_ MM	**size_t level**	Enable diagnostic output from the low-level message enqueue routines used by the MTA SDK.This item code must be followed by one additional call argument: the debug level to use.The debug level is an integer value in the range **0-20**. Enqueue diagnostics may be used to diagnose the address rewriting process, destination channel selection, header processing, and other types of processing that occurs when a message is enqueued to the MTA.Enabling this diagnostic output is equivalent to setting **DEQUEUE_ DEBUG=level** in the MTA option file.
MTA_ DEBUG_OS	None	Enable diagnostic output from the low-level operating system dependent routines used by the MTA SDK. Use of this diagnostic output is helpful when diagnosing problems associated with creating, opening, writing, or reading files. Such problems typically arise when attempting to enqueue messages to the MTA, a process that requires permissions to create and write messages in the MTA queues.Enabling this diagnostic output is equivalent to setting **OS_DEBUG=1** in the MTA option file.

Table 6–96 (Cont.) mtaInit() Item Codes, Additional Arguments, and Descriptions

Item Code	Additional Arguments	Description
MTA_ DEBUG_ SDK	None	Enable diagnostic output for the MTA SDK. When this output is enabled, diagnostic information will be output whenever the SDK returns an error result.
MTA_ ITEM_LIST	**mta_item_list_t *item_ list**	Specify a pointer to an item list array. The item list array must be terminated with a final array entry with an item code value of zero. For further information on item list usage, see Item Codes and Item Lists.
MTA_ INTERACTI VE	None	Indicate that the SDK will be used by an interactive program. In an interactive scenario, the SDK manages some of the MTA resources differently than when running as a channel program. For instance, closing the MTA log file after every completed message submission or dequeue operation.

Return Values

Table 6–97 lists the **mtaInit()** return values.

Table 6–97 mtaInit() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	A required argument to an item code was NULL.
MTA_FOPEN	Unable to initialize the MTA SDK. Unable to read one or more configuration files. Issue the following command for further information: **imsimta test -rewrite**
MTA_NO	Unable to initialize the MTA SDK. Issue the following command for further information: **imsimta test -rewrite**
MTA_NOSUCHITEM	An invalid item code was specified.

Example

For normal use:

```
mtaInit(0);
```

To select SDK diagnostics:

```
mtaInit(MTA_DEBUG_SDK, 0);
```

mtaLog()

Write diagnostic output to the channel's log file.

Syntax

```
void mtaLog(const char *fmt, ...);
```

Arguments

Table 6–98 lists the **mtaLog()** arguments.

Table 6–98 mtaLog() Arguments

Arguments	Description
fmt	Pointer to a **printf()** formatting string. The string must be NULL terminated. See your platform's C run-time library documentation for information on the formatting substitutions accepted by **printf()**.

Description

Programs that wish to write diagnostic output should use **mtaLog()** and mtaLog(). These two routines ensure that diagnostic output is directed to the same output stream as other diagnostic information generated by the MTA SDK. With one exception, consider a call to **mtaLog()** as being identical to calling the C run-time library routine **printf()**. The call arguments for the two routines are identical, including the formatting argument, **fmt**. The single exception is that, unlike **printf()**, a call to **mtaLog()** always produces a single line of output to the channel's log file. Consequently, do not attempt to write either partial or multiple lines with a single call to **mtaLog()**.

Do not include a terminating line feed or other record terminator in the output. That is, do not put a **\n** at the end of the formatting string.

A time stamp with a resolution of hundredths of a second prefaces each line of diagnostic output generated with **mtaLog()**. The time stamp uses the system clock and is reported in the local time zone.

Return Values

None

Example

```
char buf[64];

mtaLog("Version: %d.%d-%d",
       mtaVersionMajor(), mtaVersionMinor(),
       mtaVersionRevision());
mtaLog("Date/time: %s",
       mtaDateTime(buf, NULL, sizeof(buf), 0));
mtaLog("Postmaster address: %s",
       mtaPostmasterAddress(NULL, NULL));
```

The following output is generated by the preceding code example.

```
12:43:24.62: Version: 6.0-0
12:43:24.62: Date/time: Thu, 01 May 2003 12:43:24 -0700
12:43:24.63: Postmaster address: postman@mailhub.siroe.com
```

mtaLogv()

Write diagnostic output to the channel's log file.

Syntax

```
void mtaLogv(const char *fmt
             va_list     ap);
```

Arguments

Table 6–99 lists the **mtaLogv()** arguments.

Table 6–99 mtaLogv() Arguments

Arguments	Description
fmt	Pointer to a **printf()** formatting string. The string must be NULL terminated. See your platform's C run-time library documentation for information on the formatting substitutions accepted by **printf()**.
ap	A **va_list** structure as defined by the system **stdarg.h** header file.

Description

The **mtaLogv()** routine is provided for programs that either need to provide a diagnostic interface accepting a **va_list()** argument, or want to provide some generalization of **mtaLog()**. Use of **mtaLogv()** ensures that diagnostic output is directed to the same output stream as other diagnostic information generated by the MTA SDK.

With one exception, consider a call to **mtaLogv()** as being identical to calling the C run-time library routine **vprintf()**. The call arguments for the two routines are identical including the formatting argument, **fmt**. The single exception is that, unlike **vprintf()**, a call to **mtaLogv()** always produces a single line of output to the channel's log file. Consequently, do not attempt to write either partial or multiple lines with a single call to **mtaLogv()**.

Do not include a terminating line feed or other record terminator in the output. That is, do not put a **\n** at the end of the formatting string.

Return Values

None

Example

The following code fragment demonstrates a way to provide a generalization of **mtaLog()** using **mtaLogv()**.

```
#include &lt;stdarg.h>

void ourLog(our_context_t *ctx, const char *fmt, ...)
{
   char new_fmt[10240];
   va_list ap;

   /*
    * Genrate a new formatting string that includes as a prefix
    * the value of ctx-\>id then followed by the contents of the
    * supplied formatting string.
    */
   snprintf(new_fmt, sizeof(new_fmt),
            "id=%d; %s", ctx-\>id, fmt);
   va_start(ap, fmt);
   mtaLogv(new_fmt, ap);
   va_end(ap);
}
```

mtaOptionFinish()

Dispose of an option context.

Syntax

```
void mtaOptionFinish(mta_opt_t *opt_ctx);
```

Arguments

Table 6–100 lists the mtaOptionFinish() arguments.

Table 6–100 mtaOptionFinish() Arguments

Arguments	Description
opt_ctx	An option context created by **mtaOptionStart()**.

Description

Option contexts should be disposed of with a call to **mtaOptionFinish()**. The one exception to this rule are option contexts returned by **mtaDecodeMessageInfoParams()**. While those contexts may be passed to **mtaOptionFinish()**, they do not need to be because **mtaDecodeMessage()** will automatically dispose of them.

Return Values

None

Example

```
mtaOptionFinish(opt);
```

mtaOptionFloat()

Interpret and return an option's value as a floating point number.

Syntax

```
int mtaOptionFloat(mta_opt_t  *opt_ctx,
                   const char *name,
                   size_t      len,
                   double     *val);
```

Arguments

Table 6–101 lists the **mtaOptionFloat()** arguments.

Table 6–101 mtaOptionFloat() Arguments

Arguments	Description
opt_ctx	An option context created by **mtaOptionStart()**. A NULL value is permitted for this argument. When a NULL is passed, then no option value is returned.

Table 6–101 (Cont.) mtaOptionFloat() Arguments

Arguments	Description
name	Name of the option to obtain the value for. The length of this string should not exceed **ALFA_SIZE** bytes. This string must be NULL terminated if a value of zero is passed for **len**.
len	Length in bytes, not including any NULL terminator, of the option name supplied with name. If a value of zero is supplied, then the option name string must be NULL terminated.
val	Pointer to a floating point of type double to receive the option's value. If the option was not specified in the option file, then the value referenced by this pointer will be left unchanged.

Description

Use **mtaOptionFloat()** to retrieve the value of an option, interpreting its value as a floating point number. If the option is specified in the option file and its value is a valid floating point number, then its value will be returned using the **val** call argument. If the option is not specified or its value does not correctly specify a floating point number, then no value is returned and the memory pointed at by **val** is left unchanged.

The **mtaOptionFloat()}}routine can be called with a NULL value for the {{opt_ctx** argument. When this is done, **mtaOptionFloat()** immediately returns with a status code of zero and no value is returned.

This routine does not provide an indication of whether or not the option was specified in the option file. If it is important to know whether or not the option was specified, then use mtaOptionString() to test to see if the option was specified.

Return Values

Table 6–102 lists the **mtaOptionFloat()** return values.

Table 6–102 mtaOptionFloat() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_STRTRUERR	The supplied option name is too long. Its length must not exceed **ALFA_SIZE** bytes.

Example

The following code example retrieves the value of an option named **aspect_ratio**. Before calling **mtaOptionFloat()**, a default value is set for the variable to receive the value of the option. If the option was not specified in the option file, then the variable will retain that default setting. If the option was specified, then the variable will assume the value set in the file.

```
ratio = 1.0;
mtaOptionFloat(opt, "aspect_ratio", 0, &amp;ratio);
```

If it is important to know whether or not the option was specified, then use **mtaOptionString()** to test to see if the option was specified as shown in the following code example. In this example, when the routine returns, the code determines that the option was specified by whether or not the value of the **buflen** variable has changed.

```
char buf[1];
```

```
size_t buflen;

buflen = 0xffffffff;
mtaOptionString(opt, "aspect_ratio", 0, buf, &amp;buflen,
                sizeof(buf));
ratio_specified = (buflen != 0xffffffff) ? 1 : 0;
```

mtaOptionInt()

Interpret and return an option's value as an integer number.

Syntax

```
int mtaOptionInt(mta_opt_t  *opt_ctx,
                 const char *name,
                 size_t      len,
                 int        *val);
```

Arguments

Table 6–103 lists the **mtaOptionInt()** arguments.

Table 6–103 mtaOptionInt() Arguments

Arguments	Description
opt_ctx	An option context created by **mtaOptionStart()**. A NULL value is permitted for this argument. When a NULL is passed, then no option value is returned.
name	Name of the option to obtain the value for. The length of this string should not exceed **ALFA_SIZE** bytes. This string must be NULL terminated if a value of zero is passed for **len**.
len	Length in bytes, not including any NULL terminator, of the option name supplied with name. If a value of zero is supplied, then the option name string must be NULL terminated.
val	Pointer to an integer of type **int** to receive the option's value. If the option was not specified in the option file, then the value referenced by this pointer will be left unchanged.

Description

Use **mtaOptionInt()** to retrieve the value of an option, interpreting its value as an integer-valued number. If the option is specified in the option file and its value is a valid integer, then its value will be returned using the **val** call argument. If the option is not specified or its value does not correctly specify an integer, then no value is returned and the memory pointed at by **val** is left unchanged.

The routine can be called with a NULL value for the **opt_ctx** argument. When this is done, **mtaOptionInt()** immediately returns with a status code of zero and no value is returned.

This routine does not provide an indication of whether or not the option was specified in the option file. If it is important to know whether or not the option was specified, then use **mtaOptionString()** to test to see if the option was specified as shown in the code example.

Return Values

Table 6–104 lists the **mtaOptionInt()** return values.

Table 6–104 mtaOptionInt() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_STRTRUERR	The supplied option name is too long. Its length must not exceed **ALFA_SIZE** bytes.

Example

In the following code example, the value of an option named **max_blocks** is retrieved. Before calling **mtaOptionInt()**, a default value is set for the variable to receive the value of the option. If the option was not specified in the option file, then the variable will retain that default setting. If the option was specified, then the variable will assume the value set in the file.

```
blocks = 1024;
mtaOptionInt(opt, "max_blocks", 0, &amp;blocks);
```

The following code example illustrates how upon return from **mtaOptionString()**, the code determines that the option was specified by whether or not the value of the **buflen** variable has changed.

```
char buf[1];
size_t buflen;

buflen = 0xffffffff;
mtaOptionString(opt, "max_blocks", 0, buf, &amp;buflen, sizeof(buf));
blocks_specified = (buflen != 0xffffffff) ? 1 : 0;
```

mtaOptionStart()

Open, parse, and load into memory an MTA option file.

Syntax

```
int mtaOptionStart(mta_opt_t **opt_ctx,
                   const char *path,
                   size_t      len,
                   int         item_code);
```

Arguments

Table 6–105 lists the **mtaOptionStart()** arguments.

Table 6–105 mtaOptionStart() Arguments

Arguments	Description
opt_ctx	On successful return, a pointer to an option context created by **mtaOptionStart()**. This option context represents the options read from either an option file or channel-specific options read from the current configuration.
path	Optional file path to the option file to read. If not specified or of zero length, channel-specific options are read from the current configuration.

Table 6–105 (Cont.) mtaOptionStart() Arguments

Arguments	Description
len	Length in bytes, not including any NULL terminator, of the file path. This argument is ignored when a NULL is passed for **path**. When **path** is non-NULL and a value of zero is supplied for **len**, then **path** is assumed to be a NULL-terminated string and its length is determined using **strlen()**.
item_code	List of item codes with optional arguments, terminated by a 0 item code value. The supported item codes and their arguments are: ▪ **MTA_CHANNEL** Arguments: **char** **channel_name* **size_t** *channel_length* Description: Specifies the name of the channel whose channel-specific options are to be loaded from the current configuration. *channel_name* is a pointer to the channel name; *channel_length* is the length of channel name.

Description

mtaOptionStart() is used to set things up to read either MTA option files or channel-specific options from the current configuration. In the case of channel-specific options, the channel name can be specified by the **MTA_CHANNEL** item code. If **MTA_CHANNEL** isn't specified or is of zero length, the currently selected channel for the running channel program is used if one exists.

On successful return, the values of individual options can be retrieved with the routines shown in Table 6–106.

Table 6–106 mtaOptionStart() Routine Names

Routine Names	Description
mtaOptionFloat()	Retrieve the value of a floating point valued option.
mtaOptionInt()	Retrieve the value of an integer valued option.
mtaOptionString()	Retrieve the string representation of an options value.

These routines are designed such that if the requested option does not exist, then no value is returned. This allows code to assign to a variable an option's default value, then attempt to retrieve an explicitly set value from the option file. During the retrieval, the address of the variable can be passed. If the option is specified in the option file, then the value of the variable will be replaced with the value from the option file. If the option is not specified, then the default value stored in the variable is left unchanged. Code examples of such usage are provided in the individual routine descriptions.

Once finished obtaining the values of any options, unload the options from memory and dispose of the option context with **mtaOptionFinish()**.

When the underlying option file does not exist, **mtaOptionStart()** still returns a success status code. However, a NULL value is returned for the pointer to the option context. The other option routines accept a NULL value for an option context pointer and will behave as though the requested option is not specified in the option file. This behavior reflects the fact that MTA option files are considered optional. If a channel's option file does not exist, then the channel is supposed to use its default settings for its

options. This also simplifies coding, allowing programs not to have to worry about whether or not the option file exists and whether or not the option context pointer is NULL. If, however, the existence of an option file is mandatory, then a program can detect that the file does not exist by seeing if the returned value for the option context pointer is NULL as shown in the code example section that follows.

If an explicit option file path is specified with the **path** call argument, then the path can be a relative file path or an absolute file path. File paths can be prefixed with any of the symbolic MTA directory names specified in the **imta_tailor** file. For example, the entry shown in the following code fragment specifies a file named **mmsc_gateway.cnf** located in the **nmsc** subdirectory of the MTA configuration directory. Note that a colon separates the symbolic name from the remainder of the path.

```
IMTA_TABLE:/mmsc/mmsc_gateway.cnf
```

If no file path is specified, then the file specified with the **PMDF_CHANNEL_OPTION** environment variable will be opened and read. That environment variable is established by the Job Controller for the channel programs that it runs. It will always have the following format:

```
IMTA_TABLE:_channel-name__option
```

where **channel-name** is the name of the channel being run. The following example demonstrates how the environment variable settings are effected for **tcp_local** channel:

```
PMDF_CHANNEL=tcp_local
PMDF_CHANNEL_OPTION=IMTA_TABLE:tcp_local_option
```

Return Values

Table 6–107 lists the **mtaOptionStart()** return values.

Table 6–107 mtaOptionStart() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_BADARGS	A NULL value was supplied for the **opt_ctx** call argument.
MTA_FOPEN	Unable to open the option file. File access permissions are the likely cause for this error.
MTA_NO	An error occurred while reading or parsing the option file.
MTA_NOMEM	Insufficient virtual memory.
MTA_STRTRUERR	The supplied file path is too long. Its length must not exceed **ALFA_SIZE** bytes.

Example

```
opt_ctx = NULL;
if (mtaOptionStart(&amp;opt_ctx, NULL, 0, 0))
    /*
     * Error loading the option file
     */
else if (!opt_ctx)
    /*
     * Option file did not exist
     */
```

mtaOptionString()

Return an option's value as a string.

Syntax

```
int mtaOptionString(mta_opt_t  *opt_ctx,
                    const char *name,
                    size_t      len,
                    const char *str,
                    size_t     *str_len,
                    size_t      str_len_max);
```

Arguments

Table 6–108 lists the **mtaOptionString()** arguments.

Table 6–108 mtaOptionString() Arguments

Arguments	Description
opt_ctx	An option context created by **mtaOptionStart()**. A NULL value is permitted for this argument. When a NULL is passed, then no option value is returned.
name	Name of the option to obtain the value for. The length of this string should not exceed **ALFA_SIZE** bytes. This string must be NULL terminated if a value of zero is passed for **len**.
len	Length in bytes, not including any NULL terminator, of the option name supplied with name. If a value of zero is supplied, then the option name string must be NULL terminated.
str	A pointer to a buffer to receive the NULL terminated value of the specified option. The MTA allows channel options to have a maximum length of **BIGALFA_SIZE** bytes. As a result, this buffer should in general have a length of at least **BIGALFA_SIZE+1** bytes. If the option was not specified in the option file, then the contents of the buffer is left untouched.
str_len	An optional pointer to a **size_t** to receive the length in bytes of the returned option value string, **str**. A value of NULL may be passed for this call argument.
str_len_max	The maximum size in bytes of the buffer pointed at by **str**.

Description

Use **mtaOptionString()** to retrieve the string representation of an option's value. If the option is specified in the option file, then its value and length will be returned via the **str** and **str_len** call arguments. If the option is not specified then no value is returned and the memory pointed at by **str** and **str_len** are left unchanged. This routine can be called with a NULL value for the **opt_ctx** argument. When this is done, **mtaOptionString()** immediately returns with a status code of zero and no option value is returned.

Return Values

Table 6–109 lists the **mtaOptionString()** return values.

Table 6–109 mtaOptionString() Return Values

Return Values	Description
0	Normal, successful completion.
MTA_STRTRU	Supplied buffer pointed at by **buf** is too small. The returned value has been truncated to fit. Truncated value is NULL terminated. The buffer should have a length of at least **BIGALFA_SIZE+1** bytes.
MTA_STRTRUERR	The supplied option name is too long. Its length must not exceed **ALFA_SIZE** bytes.

Example

In the code example that follows, the value of an option named **mail_url** is retrieved. Before calling **mtaOptionString()**, a default value is set for the variable to receive the value of the option. If the option was not specified, then the variable will retain that default setting. If the option was specified, then the variable will assume the value set by that specification.

```
char url[1024];

strcpy(url, "mail_to:webmaster@siroe.com");
mtaOptionString(opt, "mail_url", 0, url, NULL, sizeof(url));
```

mtaPostmasterAddress()

Obtain the MTA local postmaster address.

Syntax

```
const char *mtaPostmasterAddress(const char **address,
                                 size_t      *len,
                                 int          item code, ...)
```

Arguments

Table 6–110 lists the **mtaPostmasterAddress()** arguments.

Table 6–110 mtaPostmasterAddress() Arguments

Arguments	Description
address	Optional pointer to receive the memory address of the string buffer containing the MTA local postmaster address. The string will be NULL terminated. A value of NULL may be passed for this argument.
len	Optional address of a **size_t** to receive the length in bytes of the postmaster address. A value of NULL may be passed for this argument.
item code	Reserved for future use. A value of zero (**0**) must be passed for this argument.

Description

This routine returns a pointer to a NULL terminated string containing the MTA local postmaster address. This address is suitable, for instance, for inclusion in the **From:** header line of notification messages as shown in the code example for this routine.

It is usually not a good idea for programs to send mail to the postmaster's address. In many situations, sending mail to the postmaster when failures occur can lead to mail loops if the mail sent to the postmaster itself fails, and generates a message to the postmaster, which then fails, and generates yet another message to the postmaster, and so on.

On a successful completion, the address of the string buffer containing the postmaster's address is returned using the **address** call argument. That same address is also returned as the return status.

Return Values

In the event of an error, a value of NULL is returned as the status and **mta_errno** is set with a status code indicating the underlying error. Table 6–111 lists the **mtaPostmasterAddress()** error status codes.

Table 6–111 mtaPostmasterAddress() Error Status Codes

Error Status Codes	Description
MTA_FOPEN	Unable to initialize the MTA SDK. Unable to read one or more configuration files. For further information, issue the following command: **imsimta test -rewrite**
MTA_NO	Unable to initialize the MTA SDK. For further information, issue the following command: **imsimta test -rewrite**

Example

The following example shows how to use this routine to include the postmaster address in the **From:** header line of a notification message:

```
mtaEnqueueWriteLine(nq, "From: Postmaster &lt;", 0,
                    mtaPostmasterAddress(NULL, NULL, 0), 0,
                    "\>", 0, NULL);
```

mtaStackSize()

Obtain the minimum thread stack size required when using the MTA SDK.

Syntax

```
size_t mtaStackSize(void);
```

Arguments

None

Description

A number of the run-time libraries used by the MTA SDK make intensive use of stack variables. As a result, some MTA SDK operations can require a larger than usual thread stack size. The minimum thread stack size required for typical MTA SDK operations, such as message dequeue and enqueue operations, can be obtained with **mtaStackSize()**. When writing multi-threaded code, ensure that any threads that will be calling SDK routines have a stack size at least as large as the value returned by **mtaStackSize()**. Failure to do may result in abnormal process terminations when a thread overruns its stack.

Return Values

The minimum thread stack size required for MTA SDK operations.

Example

None

mtaStrError()

Obtain a text description of an error status code.

Syntax

```
const char *mtaStrError(int code,
                        int item_code);
```

Arguments

Table 6–112 lists the **mtaStrError()** arguments.

Table 6–112 mtaStrError() Arguments

Arguments	Description
code	The MTA SDK error status to obtain a text description for.
item_code	Reserved for future use. A value of zero must be supplied for this call argument.

Description

Use **mtaStrError()** to obtain English language descriptions of MTA SDK error codes. These descriptions are intended solely for use in fine-grained diagnostic output. They are not intended for reading by end users of programs written using the MTA SDK.

Return Values

A pointer to a NULL terminated string containing the error code description.

Example

```
ires = mtaEnqueueStart(&amp;nq, from, 0, 0);
if (ires)
    printf("mtaEnqueueStart() returned %d; %s\n",
           ires, mtaStrError(ires, 0));
```

mtaUniqueString()

Generate a system-wide unique string.

Syntax

```
const char *mtaUniqueString(char   *buf,
                            size_t *len,
                            size_t  max_len);
```

Arguments

Table 6–113 lists the **mtaUniqueString()** arguments.

Table 6–113 mtaUniqueString() Arguments

Arguments	Description
buf	A pointer to a buffer to receive the NULL terminated unique string. The buffer should be at least 20 bytes long.
len	An optional pointer to a **size_t** to receive the length in bytes of the returned unique string. This length does not include the NULL terminator that terminates the returned unique string. A value of NULL can be passed for this call argument.
len_max	The maximum size in bytes of the buffer pointed at by **buf**.

Description

The **mtaUniqueString()** routine may be used to generate a system-wide unique string. The strings generated are suitable for use as MIME boundary markers and file names. On a successful completion, the unique string is stored in the buffer pointed at by the **buf** argument. Additionally, the value of the **buf** argument is returned as the routines return status.

Return Values

In the event of an error, **mtaUniqueString()** will return NULL. The error status code may be obtained by examining the value of **mta_errno**. Table 6–114 lists the **mtaUniqueString()** error status codes.

Table 6–114 mtaUniqueString() Error Status Codes

Error Status Codes	Description
MTA_BADARGS	A value of NULL was supplied for the **buf** argument.
MTA_STRTRUERR	The **buf** buffer is too small.

Example

This routine is used in "Example 5-2 Decoding MIME Messages Complex Example".

mtaVersionMajor()

Obtain the major version number associated with the MTA SDK library.

Syntax

```
int mtaVersionMajor(void);
```

Arguments

None

Description

Return the major version number associated with the MTA SDK library.

Return Values

The SDK major version number.

Example

```
printf("MTA SDK Version %d.%d-%d\n"
       mtaVersionMajor(), mtaVersionMinor(),
       mtaVersionRevision())
```

mtaVersionMinor()

Obtain the minor version number associated with the MTA SDK library.

Syntax

```
int mtaVersionMinor(void);
```

Arguments

None

Description

Return the minor version number associated with the MTA SDK library.

Return Values

The SDK minor version number.

Example

```
printf("MTA SDK Version %d.%d-%d\n"
       mtaVersionMajor(), mtaVersionMinor(),
       mtaVersionRevision());
```

mtaVersionRevision()

Obtain the revision level associated with the MTA SDK library.

Syntax

```
int mtaVersionRevision(void);
```

Arguments

None

Description

Return the revision level associated with the MTA SDK library.

Return Values

The SDK revision level.

Example

```
printf("MTA SDK Version %d.%d-%d\n"
mtaVersionMajor(), mtaVersionMinor(), mtaVersionRevision());
```

7

Using Callable Send mtaSend()

The Oracle Communications Messaging Server MTA Callable Send facility, **mtaSend()**, is a single procedure that is used to send (enqueue) mail messages of local origin; that is, to originate mail from the local host. Because the **mtaSend()** routine is not as flexible as the SDK routines and will take possibly undesirable, but necessary, authentication steps (such as, the addition of a **Sender:** header line), the MTA SDK routines should generally be used by programs that need to resend, forward, send through a gateway, or otherwise route mail messages.

The **mtaSend()** routine may be used simultaneously with the MTA SDK routines.

Sending a Message

Each message sent with **mtaSend()** must have a corresponding item list describing the message. The entries in this item list specify the message's **From:** and **To:** addresses as well as input sources for the content of the message.

The basic steps in sending a message with **mtaSend()** are:

1. Build an item list to pass to **mtaSend()**. To build an item list, complete the following steps:
   a. Specify any special processing options, such as MTA_BLANK, or MTA_IGNORE_ERRORS.
   b. Specify the message's envelope **From:** address with the MTA_USER item.
   c. Specify the message's **To:**, **Cc:**, and **Bcc:** addresses with the MTA_TO, MTA_CC, and MTA_BCC items.
   d. Specify an initial message header in one of two ways:

      - Specify an input source that supplies each of the initial message header lines (MTA_HDR_FILE MTA_HDR_PROC).

      - Specify the content of individual message header lines with individual item codes (MTA_SUBJECT, MTA_HDR_LINE).

   e. Specify the input sources for the message body with the MTA_MSG_FILE or MTA_MSG_PROC items.
   f. Terminate the item list with an item code of value 0 (MTA_END_LIST).
2. Pass the item list to **mtaSend()**.
3. Check the return status from **mtaSend()**. For a description of all item codes and their return status values, see "mtaSend() Routine Specification".

To enqueue additional messages, simply repeat these steps.

Envelope and Header From Addresses

The envelope **From:** address for a message should be specified with the MTA_USER item code. With this item code, only the local part of a mail address may be specified, that is, the user name. The **mtaSend()** routine will automatically append the official local host name to the user name so as to produce a valid mail address.

The MTA_ENV_FROMitem code may be used to explicitly specify a complete envelope **From:** address but this is usually not necessary. Applications that enqueue nonlocal mail should probably be using the SDK routines rather than **mtaSend()**.

If neither MTA_USER nor MTA_ENV_FROM are specified, then the user name associated with the current process will be used for the envelope **From:** address. When MTA_USER is used, the **From:** header line will be derived from the envelope **From:** address. When MTA_ENV_FROM is used, the **From:** header line will be derived from the user name of the current process. In either case, if a **From:** header line is supplied in an initial header, then a **Sender:** header line will be added to the message header. The initial **From:** header line will be left intact and the address specified, and **Sender:** address will be derived from either the envelope **From:** address (**MTA_USER**) or from the user name of the current process, that is, from MTA_ENV_FROM.

Only privileged users may use MTA_USER to specify a user name different than that of the current process. To be considered a "privileged" process on UNIX systems, the process must have the same (real) user ID (**UID**) as either the **root** or Messaging Server account.

To, Cc, and Bcc Addresses

The list of **To:**, **Cc:**, and **Bcc:** addresses to send a message to is built up, one address at a time, with item-list entries. Each item-list entry specifies the type of address (**To:**, **Cc:**, or **Bcc:**) and a string containing the address.

The type of address is denoted by the item code, **MTA_TO**, **MTA_CC**, or **MTA_BCC**, associated with the item-list entry. The **mtaSend()** routine uses this information to build the message envelope **To:** address list and **To:**, **Cc:**, and **Bcc:** header.

To specify an envelope-only address that should not appear in the message header (for example, an active transport address), use MTA_ENV_TO. Likewise, to specify a header-only address that should not appear in the envelope, such as, an inactive address, use MTA_HDR_TO, MTA_HDR_CC, or MTA_HDR_BCC, as appropriate.

When one or more of the **To:**, **Cc:**, or **Bcc:** addresses is illegal, the **mtaSend()** routine will not, by default, indicate which addresses were in error. However, the differentiation can be achieved by using the MTA_ADR_STATUS item code. When this item code is used, the **item_status** field associated with an address will be set either to zero (0) if the address was accepted, or to a non-zero value if there was an error processing the address.

When **item_status** is zero, **item_smessage** points to a NULL terminated string containing the rewritten form of the address. When **item_status** has a non-zero value, **item_smessage** points to a NULL terminated string containing an error message suitable for printing for diagnostic purposes.

Message Headers and Content

The body of a message, that is, the message content, is built up from zero or more input files or procedures. The input files and procedures are read or invoked in the order specified in the item list passed to the **mtaSend()** routine. The message body is

built up by appending the next input source to the end of the previous input source. A blank line will be inserted in the message as a separator between input sources if the MTA_BLANK item is requested in the item list. The MTA_MSG_FILE and MTA_MSG_PROCitem codes are used to specify the name or address of input files or procedures.

An initial message header may be supplied from either an input file or procedure. The message header will then be modified as needed when the message is enqueued. The MTA_HDR_FILE and MTA_HDR_PROC items are used to specify the name or address of an input file or procedure. If an initial message header is to be supplied, it must appear in the item list before any **MTA_MSG_FILE** or **MTA_MSG_PROC** items. A blank line must be supplied at the end of the message header, or at the start of the first message-body input source. This blank line will automatically be supplied when the **MTA_BLANK** item code is specified in the item list.

The **MTA_MODE_** and **MTA_ENC_** items control the access mode and encodings applied to message body input sources. These items set the current access mode and encoding to be applied to all subsequent input sources that appear in the item list. The default access mode is MTA_MODE_TEXT, which uses text mode access. The default encoding is MTA_ENC_UNKNOWN, which results in no encoding of the data.

The binary access mode will not be applied to input procedures. The access mode and encoding item codes do not apply to input sources for an initial message header, which is always accessed using the default access mode and never encoded.

Input procedures use the following calling format:

ssize_t proc(const char **bufadr)

where **const char **bufadr** is the address of pointer to the starting memory location of the next piece of input data.

The return value is **ssize_t**, which gives the length of the input data. A value that is equal to or greater than zero (**0**) indicates success. A value of minus one (**-1**) indicates that there is no further data to return (**EOF**). Any other negative value indicates an error for which processing should be aborted.

The procedure will be repeatedly called until a negative value is returned, which indicates all input data has been retrieved or an error occurred.

Required Privileges for mtaSend()

Like the MTA SDK routines, privileges are required in order to use **mtaSend()**. Enqueuing messages requires privileges sufficient to create, open, read from, and write to the MTA message queue directories. On UNIX, this is accomplished by having your executable program owned and run by the MTA account or, alternatively, owned by the MTA and have the **setuid** attribute set.

In order to submit mail under a user name that differs from that of the calling process, privileges are required. On UNIX platforms, the process must have the same (real) **UID** as either the **root** or Messaging Server account.

In some applications, it is important to keep strict control over when privileges are enabled and disabled. To this end, the MTA_PRIV_ENABLE_PROC and MTA_PRIV_DISABLE_PROC item codes may be used to specify the addresses of two procedures to call immediately prior to and immediately after enqueuing a message. This allows the required privileges to be enabled only when they are needed, that is, when the message is enqueued, and to remain disabled at all other times.

The **mtaSend()** routine does not use a condition handler, so if a fatal error occurs while enqueuing a message, it is up to the calling program to trap the error and, if necessary,

disable any privileges that should be disabled. These procedures, if specified, should accept no arguments and return no function result (return value).

The privileges to be enabled must either be granted to the program using **mtaSend()** (for example, the program may have been installed with privileges), or the process running the program must have the requisite privileges. The **mtaSend()** routine and the MTA do not provide these privileges.

mtaSendDispose()

For each call to **mtaSend()** where **MTA_ADR_STATUS** is used, there should be a subsequent call to **mtaSendDispose()**.

Syntax

```
void mtaSendDispose(mta_item_list_t *item_list)
```

Arguments

Table 7–1 shows the **mtaSendDispose()** arguments.

Table 7–1 mtaSendDispose() Arguments

Argument	Description
item_list	Pointer to an array with elements of type **mta_item_list_t**. This should be an array previously passed to **mtaSend()**.

Description

Each call to this routine disposes of virtual memory allocated by **mtaSend()** for returning address status information requested with the **MTA_ADR_STATUS** item code.

Return Values

None

Example

```
...
item_list[index++].item_code=MTA_ADR_STATUS;
item_list[index++].item_code=MTA_ITEM_END;
istat=mtaSend(item_list);
...
mtaSendDispose(item_list);
```

Compiling and Linking Programs

Programs that use **mtaSend()** are linked using the same steps as the MTA SDK routines. For details, see "MTA SDK Programming Considerations".

Examples of Using mtaSend()

Example programs, written in C, are provided in this section:

- Example 7-1 Send a Simple Message

The example routines shown in this section may be found in the **examples/mta/sdk** directory.

Sending a Simple Message

The program shown in "Example 7-1 Send a Simple Message" demonstrates how to send a simple message to the **root** account. The source code itself is used as the input source for the body of the message to be sent. The **From:** address associated with the message is that of the process running the program. Comments in the program example explain the sample output line they generate.

Example 7-1 Send a Simple Message

```
/* send_simple.c Send a simple message */
#include &lt;string.h\>
#include "mtasdk.h"

/* Push an entry onto the item list */
#define ITEM(item,adr) item_list[index].item_code = item;\
  item_list[index].item_address = adr;\
  item_list[index].item_length  = adr ? strlen(adr) : 0; \
  item_list[index].item_status  = 0;\
  item_list[index++].item_smessage = NULL

main ()
{
  mta_item_list_t item_list[4];
  int index = 0;

  ITEM(MTA_TO, "root"); /* Becomes the To: line in the output */
  ITEM(MTA_SUBJECT, "send_simple.c");
  ITEM(MTA_MSG_FILE, __FILE__);/* Becomes the Subject: line */
  ITEM(MTA_END_LIST, 0);
  exit(mtaSend(item_list));
}
```

Output for Example 1 Sending a Simple Message

```
Date: 04 Oct 1992 22:24:07 -0700 (PDT)
From: jdoe@sesta.com
Subject: send_simple.c
To: root@sesta.com
Message-id: &lt;01GPKF10JIB89LV1WX@sesta.com\>
MIME-version: 1.0
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT

/* send_simple.c -- Send a simple message */
#include &lt;string.h\>
#include "mtasdk.h"

...
```

Example 2 Specifying an Initial Message Header

The program shown in "Example 7-2 Specifying an Initial Message Header" illustrates the use of the **MTA_HDRMSG_FILE** and **MTA_HDR_ADRS** item codes to enqueue a message that has already been composed, including the headers, and stored in a file. The input file is given in the "Input File for Example 2 Specifying an Initial Message Header". The resulting message is shown in "Output for Example 2 Specifying an Initial Message Header".

When the entire message, header and body, is contained in a single file, use the **MTA_HDRMSG_FILE** item code in place of the **MTA_HDR_FILE** and **MTA_MSG_FILE** item codes.

Example 7-2 Specifying an Initial Message Header

```
/* send_header.c -- Send a message with initial header */
#include &lt;string.h\>
#include "mtasdk.h"

/* Push an entry onto the item list */
#define ITEM(item,adr) item_list[index].item_code = item;\
  item_list[index].item_address = adr;\
  item_list[index].item_length  = adr ? strlen(adr) : 0;\
  item_list[index].item_status  = 0;\
  item_list[index++].item_smessage = NULL

main ()
{
  MTA_item_list_t item_list[3];
  int index = 0;

  ITEM(MTA_HDR_ADRS, 0);
  ITEM(MTA_HDRMSG_FILE, "send_header.txt");
  ITEM(MTA_END_LIST, 0);
  exit(mtaSend(item_list));
}
```

Input File for Example 2 Specifying an Initial Message Header

```
Subject: MTA SDK callable Send example
To: root@sesta.com
MIME-version: 1.0
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT
Comments: Ignore this message -- it&rsquo;s just a test

This is a test of the emergency broadcasting system!

12345678901234567890123456789012345678901234567890123456789012345678901234
5678901234567890

00000000011111111112222222222333333333344444444445555555555666666
6666677777777778
```

Output for Example 2 Specifying an Initial Message Header

```
Date: 04 Jan 2003 22:42:25 -0800 (PST)
From: system@sesta.com
Subject: MTA SDK callable Send example
To: system@sesta.com
Message-id: &lt;01GPKFNPUQF89LV1WX@sesta.com\>
MIME-version: 1.0
```

```
Content-type: TEXT/PLAIN; CHARSET=US-ASCII
Content-transfer-encoding: 7BIT
Comments: Ignore this message -- it's just a test

This is a test of the emergency broadcasting system!

123456789012345678901234567890123456789012345678901234567890123456789012345
5678901234567890

000000000111111111122222222223333333333444444444455555555556666666666677777
6666677777777778
```

Example 3 Sending a Message to Multiple Recipients

The program in "Example 7-3 Sending a Message to Multiple Recipients" demonstrates the following points:

- Sending a message to multiple recipients.
- Obtaining the status (legal, illegal) of each envelope recipient address (that is, active transport address).

The message is sent to one **To:** address, a **Cc:** address, and a **Bcc:** address. After **mtaSend()** is called, any status message associated with each address is displayed.

The log output produced by running the program is shown in "Output for Example 3 Sending a Message to Multiple Recipients".

The following items of note are identified in the comments in the program:

- Instruct **mtaSend()** to return a status message for each envelope recipient address.
- Specify some **To:**, **Cc:**, and **Bcc:** addresses.
- Send the message.
- Display any returned status messages.

Example 7-3 Sending a Message to Multiple Recipients

```
/* send_multi.c -- Send a message to multiple recipients */
#include &lt;stdio.h\>
#include &lt;string.h\>
#include "mtasdk.h"

#define ITEM(item,adr) item_list[index].item_code = item;\
  item_list[index].item_address = adr;\
  item_list[index].item_length  = adr ? strlen(adr) : 0;\
  item_list[index].item_status  = 0;\
  item_list[index++].item_smessage = NULL

main ()
{
  int index = 0, istat, i;
  mta_item_list_t item_list[7];

  /* Specify the Subject: header line and message input source */
  ITEM(MTA_SUBJECT, "send_multi.c");
  ITEM(MTA_MSG_FILE, __FILE__);

  /* Return per address status/error messages */
  ITEM(MTA_ADR_STATUS, 0); /* Instructs mtaSend() to return a */
                           /* status message for each envelope */
                           /* recipient address                */
```

```
  /* Specify regular Bcc:, To:, and Cc: addresses */
  ITEM(MTA_BCC, "root");
  ITEM(MTA_TO, "abuse@sample.com");
  ITEM(MTA_CC, "postmaster@sample.com");

  /* Now terminate the item list */
  ITEM(MTA_END_LIST, 0);

  /* And send the message */
  istat = mtaSend(item_list);/* Sends the message. */

  /* Display the address status messages provided that no */
  /* error other than MTA_HOST has occurred               */



  for (i = 0; i &lt; index; i++) /* Display any returned status */
                              /* messages                    */
       if (item_list[i].item_smessage)
           printf ("%s: %s - %s\n",
                   (const char *)item_list[i].item_address,
                   item_list[i].item_status ? "Failed" :
                                              "Succeeded",
                   item_list[i].item_smessage);

  /* Dispose of status messages */
  mtaSendDispose(item_list);
  exit(istat);
}
```

Output for Example 3 Sending a Message to Multiple Recipients

```
Succeeded: root@sample.com
Succeeded: abuse@sample.com
Succeeded: postmaster@sample.com
```

Example 4 Using an Input Procedure to Generate the Message Body

The program shown in "Example 7-4 Using an Input Procedure to Generate the Message Body" uses an input procedure as the source for the body of a message to be sent. In the program, the input procedure **msg_proc** will read input until the runtime library routine **fgets()** signals an EOF condition, for example, a **control-D** has been input. The address of the procedure **msg_proc** is passed to **mtaSend()** using a **MTA_MSG_PROC** item code. The **mtaSend()** routine repeatedly calls the **msg_proc** procedure, until a negative value is returned by the procedure.

Example 7-4 Using an Input Procedure to Generate the Message Body

```
/* send_input.c -- Demonstrate the use of MTA_MSG_PROC */
#include &lt;stdio.h\>
#include &lt;stdlib.h\>
#include &lt;string.h\>
#include "mtasdk.h"
#ifdef _WIN32
typedef long ssize_t;
#endif

/* Push an entry onto the item list */
#define ITEM(item,adr) item_list[index].item_code = item;\
item_list[index].item_address = adr;\
```

```
item_list[index].item_length  = 0;\
item_list[index].item_status  = 0;\
item_list[index++].item_smessage = NULL

ssize_t msg_proc(const char **bufadr)
{
  static char buf[1024];

  if (!bufadr)
      return(-2); /* Call error; abort */

  printf("input: ");
  if (fgets(buf, sizeof(buf), stdin))
  {
    *bufadr = buf;
    buflen = strlen(buf);
    if (buf[buflen-1] == &rsquo;\n&rsquo;)
      buflen -= 1;
    return(buflen);
  }
  else
    return(-1);  /* EOF */
}

main ()
{
  int istat, index = 0;
  mta_item_list_t item_list[4];

  STRITEM(MTA_SUBJECT, "send_input.c");
  STRITEM(MTA_TO, "root");
  ITEM(MTA_MSG_PROC, msg_proc);
  ITEM(MTA_END_LIST, 0);
  exit(mtaSend(item_list));
}
```

8

mtaSend() Routine Specification

This chapter contains the functional specification of the **mtaSend()** routine.

List of Item Codes

- MTA_ADR_NOSTATUS
- MTA_ADR_STATUS
- MTA_BCC
- MTA_BLANK
- MTA_CC
- MTA_CHANNEL
- MTA_CFILENAME
- MTA_CFILENAME_NONE
- MTA_CTYPE
- MTA_ENC_BASE64
- MTA_ENC_BASE85
- MTA_ENC_BINHEX
- MTA_ENC_BTOA
- MTA_ENC_COMPRESSED_BASE64
- MTA_ENC_COMPRESSED_BINARY
- MTA_ENC_COMPRESSED_UUENCODE
- MTA_ENC_HEXADECIMAL
- MTA_ENC_NONE
- MTA_ENC_PATHWORKS
- MTA_ENC_QUOTED_PRINTABLE
- MTA_ENC_UNKNOWN
- MTA_ENC_UUENCODE
- MTA_END_LIST
- MTA_ENV_FROM
- MTA_ENV_TO

- MTA_FRAGMENT_BLOCKS
- MTA_FRAGMENT_LINES
- MTA_FROM
- MTA_HDR_ADRS
- MTA_HDR_BCC
- MTA_HDR_CC
- MTA_HDR_FILE
- MTA_HDR_LINE
- MTA_HDR_NOADRS
- MTA_HDR_NORESENT
- MTA_HDR_PROC
- MTA_HDR_RESENT
- MTA_HDR_TO
- MTA_HDRMSG_FILE
- MTA_HDRMSG_PROC
- MTA_IGNORE_ERRORS
- MTA_INTERACTIVE
- MTA_ITEM_LIST
- MTA_MAX_TO
- MTA_MODE_BINARY
- MTA_MODE_TEXT
- MTA_MSG_FILE
- MTA_MSG_PROC
- MTA_NOBLANK
- MTA_NOIGNORE_ERRORS
- MTA_PRIV_DISABLE_PROC
- MTA_PRIV_ENABLE_PROC
- MTA_SUBADDRESS
- MTA_SUBJECT
- MTA_TO
- MTA_USER

mtaSend() Syntax

int mtaSend(mta_item_list_t *item_list)

Arguments

item_list

The **mtaSend()** routine takes only one argument, **item_list**, which is a pointer to an array of item descriptors. Each item descriptor specifies an action to be taken, and provides the information needed to perform that action.

The list of item descriptors is terminated with an entry containing the **MTA_END_LIST (0)** item code.

Each item descriptor has the following C-style structure declaration:

```
struct {
  int         item_code;
  const void *item_address;
  int         item_length;
  int         item_status;
  const char *item_smessage;
} mta_item_list_t;
```

Item Descriptor Fields

item_code

Integer item code specifying an action to be taken by **mtaSend()**. The include file described in "MTA SDK Concepts and Overview" defines these codes. Each item code is described later in this chapter, starting at "Item Codes".

item_address

The caller-supplied address of data to be used in conjunction with the action specified by the **item_code** field. Not all actions require that an **item_address** be supplied.

item_length

When the item code has an associated string value, this field optionally provides the length in bytes of the string, not including any NULL terminator. If a value of zero **(0)** is supplied, then the string pointed to by **item_address** must be NULL terminated, so that **mtaSend()** can automatically determine the string's length.

When the item code has an associated integer value, this field supplies that value.

item_status

When the item code **MTA_ADR_STATUS** is specified, this field will contain processing status for the associated envelope recipient address.

item_smessage

When the item code **MTA_ADR_STATUS** is specified, this field will contain the rewritten form of the envelope recipient address when the returned value of **item_status** is zero, or a textual error message when the returned value of **item_status** is non-zero.

Description

Use **mtaSend()** to send a message. The routine performs the processing carried out to address the message, generate the message's header and body, and enqueue the message as specified with the **item_list** argument. For instructions on how to use **mtaSend()**, see "Using Callable Send mtaSend()".

Item Codes

This section describes **MTA_*** item codes.

MTA_ADR_NOSTATUS

Do not return status messages for **To:**, **Cc:**, and **Bcc:** addresses. This is the default setting.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_ADR_STATUS

Return textual status messages for each envelope recipient address (that is, an active transport address) specified with any of these item codes: **MTA_TO**, **MTA_CC**, **MTA_BCC**, **MTA_HDR_TO**, **MTA_HDR_CC**, or **MTA_HDR_BCC**. When a recipient address is successfully processed, the value of the associated **item_status** field will be zero and **item_smessage** will be a pointer to a NULL terminated string containing the rewritten form of the address. When a recipient address fails to be processed successfully, the value of the associated **item_status** field will be non-zero and **item_smessage** will be a pointer to a NULL terminated error message string.

After calling **mtaSend()** with **MTA_ADR_STATUS**, call the **mtaSendDispose()** function to dispose of any dynamic memory allocated by **mtaSend()**.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_BCC

Specify a blind carbon copy (**Bcc:** address. The **item_address** and **item_length** fields specify the address and length of a string containing a **Bcc:** address. The length of the address may not exceed ALFA_SIZE bytes.

MTA_BCC is used to specify a **Bcc:** address that should appear in both the message's header and envelope.

MTA_BLANK

When processing multiple input sources, insert a blank line between the input from each source. Ordinarily, the input files are appended one after the other with no delimiters or separators. This is the action selected with the **MTA_NOBLANK** item code. By specifying the **MTA_BLANK** action, **mtaSend()** inserts a blank line between each input file. This is especially useful when the first input file is to be treated as a source of header information and the second as the message body or part thereof. This produces the requisite blank line between the message header and body.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_CC

Specify a carbon copy (**Cc:**) address. The **item_address** and **item_length** fields specify the address and length of a string containing a **Cc:** address. The length of the address may not exceed ALFA_SIZE bytes.

MTA_CC is used to specify a **Cc:** address that should appear in both the message's header and envelope.

MTA_CHANNEL

Specify the channel to act as, when enqueuing the message. If not specified, then mail will be enqueued as though sent from the local, **l**, channel. The **item_address** and **item_length** fields specify the address and length of a text string containing the name

of the channel to act as. The length of the string may not exceed CHANLENGTH bytes.

MTA_CFILENAME

When **MTA_CFILENAME** is specified, the name of the message input file will be included as a parameter in the MIME **Content-type:** header line. This action, when specified, will hold for all subsequent input files until an **MTA_CFILENAME_NONE** action is seen in the same item list.

MTA_FILENAME_NONE is the default.

MTA_CFILENAME_NONE

The default action for including or not including the name of the message input file as a parameter in the MIME **Content-type:** header line. This item code specifies that no input file is to be included.

When **MTA_CFILENAME** has been specified, it will hold for all subsequent input files until an **MTA_CFILENAME_NONE** action is seen in the same item list.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_CTYPE

Specify the body of a **Content-type:** header line. The **item_address** and **item_length** fields specify the address and length of a text string to place in the body of a **Content-type:** header line. The length of the string may not exceed ALFA_SIZE bytes. Only one **Content-type:** body may be specified.

MTA_ENC_BASE64

Encode data from all subsequent input sources using MIME's BASE64 encoding. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN.** The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_BASE85

Encode data from all subsequent input sources using Adobe's ASCII85 encoding (BASE85). This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN.** The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_BINHEX

Encode data from all subsequent input sources using the BINHEX encoding. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN.** The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_BTOA

Encode data from all subsequent input sources using the UNIX binary-to-ASCII (BTOA) encoding. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_COMPRESSED_BASE64

Encodes data from all subsequent input sources using MIME's BASE64 encoding after first compressing it using Gnu zip. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_COMPRESSED_BINARY

Compress the data with Gnu zip. No other encoding of the data will be done. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_COMPRESSED_UUENCODE

Encode data from all subsequent input sources using UUENCODE, after first compressing the data with Gnu zip. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_HEXADECIMAL

Encode data from all subsequent input sources using a hexadecimal encoding. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_NONE

Data from all subsequent input sources is left unencoded (that is, not encoded). This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_PATHWORKS

Encodes multipart and binary message contents using the OpenVMS Pathworks format. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_QUOTED_PRINTABLE

Encode data from all subsequent input sources using MIME's quoted printable encoding. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_UNKNOWN

Data from all subsequent input sources is left unencoded (that is, not encoded). This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENC_UUENCODE

Encode data from all subsequent input sources using UUENCODE. This setting may be changed with any of the other **MTA_ENC_** item codes. The default encoding is **MTA_ENC_UNKNOWN**. The **item_address** and **item_length** fields are ignored for this item code.

MTA_END_LIST

Terminate an item list. This item code, when encountered, signals the end of the item list. The **item_address** and **item_length** fields are ignored for this item code.

MTA_ENV_FROM

Specify the envelope **From:** address to associate with a message. The **item_address** and **item_length** fields specify the address and length of a text string containing the envelope **From:** address to use for the message submission. The length of the string may not exceed ALFA_SIZE bytes. Only one envelope **From:** address may be specified.

The **MTA_ENV_FROM** action should be used when the envelope **From:** address is not a local address. When the address is a local address, then only the user name should be specified using the **MTA_USER** action.

If this action and the **MTA_USER** actions are not specified, then the user name associated with the current process will be used.

Do not use this item code in conjunction with the **MTA_USER** or **MTA_SUB_USER** item codes.

MTA_ENV_TO

Specify an envelope-only **To:** address (that is, an active recipient), which should not appear in the message's header. The **item_address** and **item_length** fields specify the address and length of a string containing a **To:** address. The length of the address may not exceed ALFA_SIZE bytes.

MTA_FRAGMENT_BLOCKS

Specify the maximum number of blocks per message. If, when the message is enqueued, the message size exceeds this limit, then the message will be fragmented into smaller messages, each fragment no larger than the specified block size. The individual fragments are MIME compliant messages that use MIME's **message/partial** content type. MIME compliant mailers or user agents that receive the fragments may automatically reassemble the fragmented message. (MTA channels must be marked with the **defragment** keyword in order for automatic message reassembly to occur.)

The size of a block may vary from site to site. Sites can change this value from its default value of 1,024 bytes. Use the MTA SDK routine **mtaBLOCK_SIZE** to determine the size in bytes of a block.

The **item_length** field specifies the maximum block size per message or message fragment. By default, no limit is imposed.

MTA_FRAGMENT_LINES

Specify the maximum number of message lines per message. If, when the message is enqueued, the number of message lines exceeds this limit, then the message will be fragmented into smaller messages, each fragment with no more than the specified

number of lines. The individual fragments are MIME compliant messages that use MIME's **message/partial** content type. MIME compliant mailers or user agents that receive the fragments may automatically reassemble the fragmented message. (MTA channels must be marked with the **defragment** keyword in order for automatic message reassembly to occur.)

The **item_length** field specifies the maximum number of message lines per message or message fragment. By default, no limit is imposed.

MTA_FROM

Specify the address to use in the message header's **From:** header line. The **item_address** and **item_length** fields specify the address and length of a text string containing the **From:** address. The length of the string may not exceed ALFA_SIZE bytes. Only one **From:** address may be specified.

If this action is not used, then the **From:** header line will be derived from the envelope **From:** address.

MTA_HDR_ADRS

Specify **MTA_HDR_ADRS** to request that the message also be sent to recipient addresses found in any input header files. The **item_address** and **item_length** fields are ignored for this item code.

MTA_HDR_BCC

Specify a header-only **Bcc:** address (that is, an inactive recipient), which should only appear in the message's header. The **item_address** and **item_length** fields specify the address and length of a string containing a **Bcc:** address. The length of the address may not exceed ALFA_SIZE bytes.

MTA_HDR_CC

Specify a header-only carbon copy (**Cc:**) address (that is, an inactive recipient), which should only appear in the message's header. The **item_address** and **item_length** fields specify the address and length of a string containing a **Cc:** address. The length of the address may not exceed ALFA_SIZE bytes.

MTA_HDR_FILE

Specify the name of an input file containing message header lines. The first input file may be a file containing a message header. In this case, it should be specified using this item code rather than **MTA_MSG_FILE**. This will ensure that the input file receives the proper processing (such as, is not encoded, accessed using text mode access). The **mtaSend()** routine uses the header lines from the input file to form an initial message header. This initial header is then modified as necessary. This functionality is useful when forwarding mail.

Note that any recipient addresses in the header file will be ignored unless **MTA_HDR_ADRS** is also specified.

The **item_address** and **item_length** fields specify the address and length of a text string containing the input file's name. The length of the string may not exceed ALFA_SIZE bytes.

MTA_HDR_LINE

Specify an additional header line to include in the message header. The **item_address** and **item_length** fields specify the address and length of the header line (field name and body) to place in the message header. The length of the string may not exceed ALFA_SIZE bytes. Any number of header lines may be added. Use one item list entry per header line.

MTA_HDR_NOADRS

Recipient addresses must be explicitly specified and any addresses in an input header file will be ignored (but will still appear in the message header). The **item_address** and **item_length** fields are ignored for this item code.

This is the default action for recipient addresses found in input header files.

MTA_HDR_NORESENT

Specify **MTA_HDR_NORESENT** to cause additional addresses to be added to existing header lines rather than through the introduction of Resent- header lines.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_HDR_PROC

Specify the address of a procedure that will return, one line at a time, header lines for the message header. The **item_address** field specifies the address of the procedure to invoke. The **item_length** field is ignored.

The calling format that must be used by the procedure is given in "Message Headers and Content".

MTA_HDR_RESENT

The **MTA_HDR_RESENT** action selects the default behavior whereby **Resent-** header lines are added as necessary to the message header when the associated header line appears in any input header files. For instance, a **Resent-to:** header line will be added if a **To:** header line already appears. The **item_address** and **item_length** fields are ignored for this item code.

MTA_HDR_TO

Specify a header-only **To:** address (that is, an inactive recipient), which should only appear in the message's header. The **item_address** and **item_length** fields specify the address and length of a string containing a **To:** address. The length of the address may not exceed ALFA_SIZE bytes.

MTA_HDRMSG_FILE

Specify the name of an input file containing both the message header and message body. The content of the file represents an RFC 2822 formatted message with at least one blank line separating the RFC 2822 header from the message body. The **mtaSend()** routine uses the header lines from the input file to form an initial message header. This initial header is then modified as necessary.

The **item_address** and **item_length** fields specify the address and length of a text string containing the input file's name. The length of the string may not exceed ALFA_SIZE bytes.

MTA_HDRMSG_PROC

Specify the address of a procedure that will return, one line at a time, each line of an RFC 822 formatted message. The RFC 822 header must come first, followed by at least one blank line, followed by the message body. The **item_address** field specifies the address of the procedure to invoke. The calling format that must be used by the procedure is given in "Message Headers and Content".

MTA_IGNORE_ERRORS

Send the message as long as at least one **To:** address was okay and at least one input source was okay. By default, the message will not be sent if any of the **To:** addresses are illegal (such as, bad syntax, restricted, unknown host), or if any of the input sources proved to be bad (such as, could not open an input file). The **item_address** and **item_length** fields are ignored for this item code.

MTA_INTERACTIVE

Do not ignore user-to-channel access checks when enqueuing mail. This should, in general, be used by programs such as user agents that enqueue mail for users.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_ITEM_LIST

The **mtaSend()** routine immediately begins processing the list of item descriptors pointed at by **item_address**. This new list will be used immediately; any remaining items in the current list will be ignored.

The **item_length** field is ignored for this item code.

MTA_MAX_TO

Specify the maximum number of envelope **To:** addresses per message copy. If, when the message is enqueued, the number of envelope **To:** addresses for the message exceeds this limit, then the message will be broken into multiple copies, each copy with no more than the specified number of envelope **To:** addresses.

The **item_length** field specifies the maximum number of envelope **To:** addresses per message copy. By default, no limit is imposed.

MTA_MODE_BINARY

Read subsequent input files as raw binary files. This setting may be changed with the **MTA_MODE_TEXT** item code. The default access mode is **MTA_MODE_TEXT**.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_MODE_TEXT

Read subsequent input files as record-oriented text files. This setting may be changed with the **MTA_MODE_BINARY** item code. The default access mode is **MTA_MODE_TEXT**.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_MSG_FILE

Specify an input file to read and include in the message body. The file will be read using the current access mode and encoded using the current encoding as specified by **MTA_MODE_** and **MTA_ENC_** item codes.

The **item_address** and **item_length** fields specify the address and length of a text string containing the name of the input file. The length of the string may not exceed ALFA_SIZE bytes.

MTA_MSG_PROC

Specify the address of a procedure that will return, one line at a time, data for the message body. Each line of input obtained from the procedure will be treated using the current access mode and encoded using the current encoding as specified by **MTA_MODE_** and **MTA_ENC_** item codes. Note, however, that the block access mode will not be applied to input procedures.

The **item_address** field specifies the address of the procedure to invoke. The **item_length** field is ignored.

The calling format that must be used by the procedure is given in "Message Headers and Content".

MTA_NOBLANK

When processing multiple input source, do not insert a blank line between the input from one source and the next. This is the default behavior. The input from each input source is appended one after the other with no delimiters or separators marking the transition between sources.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_NOIGNORE_ERRORS

Send the message only if all **To:** addresses are okay and all input sources are okay. This is the default.

The **item_address** and **item_length** fields are ignored for this item code.

MTA_PRIV_DISABLE_PROC

The address of a procedure to invoke immediately after enqueuing a message so as to disable process privileges. See the description of **MTA_PRIV_ENABLE_PROC** for details on the use of this item code.

This item code must be used in conjunction with **MTA_PRIV_ENABLE_PROC** item.

The **item_length** field is ignored for this item code.

MTA_PRIV_ENABLE_PROC

The address of a procedure to invoke immediately before enqueuing a message so as to enable process privileges.

Privileges are required to enqueue messages. It is possible to provide **mtaSend()** with the address of two procedures to call. One procedure is called immediately prior to enqueuing a message thereby allowing process privileges to be enabled. The second procedure is then called immediately after the message has been enqueued thereby allowing process privileges to be disabled.

For further details on the use of this item code, see "Required Privileges for mtaSend()".

This item code must be used in conjunction with **MTA_PRIV_DISABLE_PROC**.

The **item_length** field is ignored for this item code.

MTA_SUBADDRESS

Specify a subaddress to use when generating a return address from a user name specified with the **MTA_USER** item code. The **item_address** and **item_length** fields specify the address and length of a text string containing the subaddress. The length of the string may not exceed ALFA_SIZE bytes. Only one subaddress may be specified per message.

The **MTA_USER** action must be used in conjunction with this item code.

MTA_SUBJECT

Specify the body of a **Subject:** header line. The **item_address** and **item_length** fields specify the address and length of a text string to place in the body of a **Subject:** header line. The length of the string may not exceed ALFA_SIZE bytes. Only one **Subject:** body may be specified.

MTA_TO

Specify a **To:** address that should appear in both the message's header and envelope. The **item_address** and **item_length** fields specify the address and length of a string containing a **To:** address. The length of the address may not exceed ALFA_SIZE bytes.

MTA_USER

Specify the user name to use for the envelope **From:** and header line **From:** addresses. The **item_address** and **item_length** fields specify the address and length of a text string containing the user name.

Use this item code when the envelope **From:** address is a local address.

If the envelope **From:** address is not a local address, then the MTA_ENV_FROM action should be used.

If this action and the **MTA_ENV_FROM** actions are not specified, then the user name associated with the current process will be used.

On UNIX, the process must have the same (real) **UID** as the **root** or **mta** account. If the process lacks sufficient privileges, the **MTA_ACCESS** error will be returned.

Do not use this item code in conjunction with the **MTA_ENV_FROM** item code.

9

Error Status Codes Summary

This chapter describes the error status codes returned by the MTA SDK and **mtaSend()**.

Error Status Codes

Table 9–1 lists the error status codes, with a generic interpretation of each. For usage-specific interpretations, refer to the specific MTA SDK routine descriptions in "MTA SDK Routines", and the **mtaSend()** item code descriptions in "mtaSend() Routine Specification".

Table 9–1 MTA SDK Error Status Codes

Return Code	Numeric Value	Description
MTA_OK	**0**	Normal, successful completion.
MTA_ ACCESS	**1**	This error typically indicates that a site-supplied access mapping table has refused an envelope recipient address with a permanent error. These access mapping tables include: **SEND_ACCESS**, **ORIG_SEND_ACCESS**, **MAIL_ ACCESS**, and **ORIG_MAIL_ACCESS**.This error may also result when a mailing list has access controls which do not allow the attempted message submission to the list.
MTA_AGAIN	**2**	A temporary processing error has occurred. A number of conditions may generate this error including connectivity problems to LDAP servers, virus scanners, spam scanners, as well as quota problems.When the error is the result of an attempt to add an envelope recipient address or to complete a message enqueue, additional information may be obtained by either enabling SDK diagnostics with **mtaDebug()** or using the **MTA_REASON** item code of **mtaEnqueueTo()** or **mtaEnqueueFinish()**. In the case of **mtaEnqueueTo()**, **mtaEnqueueError()** may also be used to obtain the extended information returned with the **MTA_ REASON** item code.
MTA_ BADARGS	**3**	Bad call arguments supplied to the called routine. Typically, this will be the result of passing an invalid context or a NULL value for a required parameter.
MTA_EOF	**4**	End of data reached. When returned by **mtaDequeueLineNext()** or **mtaDequeueRecipientNext()**, this value does not indicate an error, but rather that there are, respectively, no more message lines or recipients to return.

Table 9–1 (Cont.) MTA SDK Error Status Codes

Return Code	Numeric Value	Description
MTA_ FCREATE	**5**	Unable to create a disk file. Typically, this will be the result of insufficient disk space, insufficient access rights to the channel queue directories, or a file system error of some sort. The MTA SDK creates both temporary files and message files in the channel queue directories. The temporary files result when a message being submitted exceeds in size the value of the MTA option: **MAX_ INTERNAL_BLOCKS**.
MTA_FIO	**6**	An error occurred while writing to a disk file. Typically, this will be the result of insufficient disk space or a file system error. This error is only reported when writing message files, either temporary files, or writing them in the channel queue directories.
MTA_OPEN	**7**	An error occurred while attempting to open a disk file. In regards to channel option files, this indicates that the channel option file exists but cannot be opened. Usually this is caused by insufficient access rights or a file system error.This error may also be returned when the MTA SDK is initialized and an MTA configuration file cannot be opened. Again, this usually indicates a problem with permissions or the file system. Use the **imsimta test -rewrite** utility to obtain additional diagnostic information. That utility often reports the name of the underlying configuration file associated with the error.
MTA_ NETWORK	**8**	A network read or write error has occurred. This error is associated with message dequeue processing and indicates that a communication error has occurred while attempting to contact or exchange information with the MTA Job Controller. Ensure that the Job Controller is running.
MTA_NO	**9**	Generic error message. This error message is issued in a variety of situations. In all cases, it indicates that the attempted call has failed. Consult the routine's description for an interpretation specific to the called routine. Also, consider enabling MTA SDK diagnostics with **mtaDebug()**.
MTA_ NOMEM	**10**	Insufficient virtual memory; cannot perform the requested operation.
MTA_NOOP	**11**	This error code is not presently used by the MTA SDK. In general, it is used to indicate that the requested operation was completed by doing nothing (for example, a message enqueued to zero envelope recipients is simply deleted).
MTA_ NOSUCHCH AN	**12**	The specified channel name does not exist in the MTA configuration. The channel name may have been specified explicitly with a supplied call argument or implicitly with the **PMDF_CHANNEL** environment variable.
MTA_ NOSUCHHO ST	**13**	The MTA configuration lacks the necessary information to route the specified envelope recipient address. This error typically comes up when an unrecognized, top-level domain name is used. As such, this usually indicates a syntactically valid recipient address which specifies an invalid top-level domain name (for example, **sue@siroe.siroe**). Other addressing errors, including syntax errors, may elicit this status code.
MTA_ NOSUCHITE M	**14**	An invalid item code was supplied. Either the supplied item code value does not represent a known item code or it is not an item code supported by the called routine.

Table 9–1 (Cont.) MTA SDK Error Status Codes

Return Code	Numeric Value	Description
MTA_ORDER	**15**	Routine called out of order. For example, an attempt to read the text of a queued message file was made before first reading the message's entire recipient list. Or, an attempt was made to write the content of a message being submitted before first specifying the message's recipients. Refer to the call order diagrams in for further details.
MTA_SIZE	**16**	The message being submitted cannot be enqueued: its size exceeds a site-configured size limit. Such limits are configured with a variety of options, including the MTA options **BLOCK_LIMIT** and **LINE_LIMIT**, as well as the channel options **blocklimit** and **linelimit**.
MTA_ STRTRU	**17**	The supplied buffer was not large enough to receive the result string. The result string was truncated to fit. The result string is nonetheless NULL terminated.
MTA_ STRTRUERR	**18**	The supplied buffer was not larger enough to receive the result string. Truncating the result is not meaningful or has potential for causing problems or both. Alternatively, a supplied string was too long.
MTA_ THREAD	**19**	Threading error detected. Specifically, the MTA SDK detected the simultaneous use of a single SDK context by two or more processing threads. This is not permitted.
MTA_ TIMEDOUT	**20**	This error code is not presently used by the MTA SDK. In general, it is used to indicate a timeout related error.