# Eudora Extended Message Services API Version 4

September 8, 1998

QUALCOMM Inc.

Laurence Lundblade, Julia Blumin, Scott Manjourides, Joshua Stephens

For more information write to <eudora-emsapi@qualcomm.com>

QUALCOMM Incorporated

6455 Lusk Blvd.

San Diego, CA 92121-2779

USA

# Table of Contents

# 1. Introduction

*Note: sections one through seven of this document provide overview, background and implementation guidelines for the EMS API. Detailed reference information for implementation begins in section eight.*

The Eudora Extended Message Services API (EMS API) is designed so that third party plug-ins can be added to Eudora by the end user. Plug-ins may be supplied by QUALCOMM Incorporated, an independent vendor, be available as shareware, or be authored by the end user. Plug-ins may perform transformations on e-mail messages as they are received, as they are sent or on the command of the user. Additionally, they can add attachments to messages as well as be simply a hook to another application. The API is general enough to accommodate transformations ranging from compression/decompression, to file format conversions, graphic format conversions, human language translation, digital signing and others. **U.S. developers of plug-ins which perform encryption/decryption should contact the U.S. Department of State's Office of Defense Trade Controls in order to determine the licensing requirements applicable to exports of such translator plug-ins from the United States.**

When Eudora starts up it will search for plug-ins on the user's system. It will look for Windows DLL's or Macintosh Components in a set of specific places on the user's system. Once located, the plug-ins will show up as menu items and check boxes in the Eudora user interface and/or be invoked automatically as messages are sent and received. When invoked, plug-ins may interact directly with the user by putting up their own dialogue boxes and menu items for attachments and tools.

Each plug-in may contain *translators, attachers and special tool menus* in it.

A *translator* performs some transformation on a message. It is often convenient to put several translators in one plug-in because they may share a lot of code or other resources. It is also possible for a translator to be used simply as a hook for access to messages as they are received, viewed or sent. That is, a translator may perform no translation at all.

## *1.1. Translators*

The translators in a plug-in are executed in the following *contexts*:

**On-arrival** — When the message arrives from the mail server non-interactive translations can occur. It is also possible for a translator to indicate processing (MIME parsing and further translation) of the message structure should be suspended until it can be done in an interactive context.

**On-display** — When the message is selected for display an interactive translator may be automatically run. The result will be displayed to the user.

**On-request** — Translations for both received messages and messages under composition can be selected from a menu item. The translation will be performed right away and the result shown to the user.

**Q4-transmission** — More properly described as "queue and call on transmission," this translation is selected by icon from the top bar of the message composition window. Then when the message is actually being transmitted to the SMTP server the translation is performed.

**Q4-completion** — More properly described as "queue and call on completion," this translation is selected by icon from the top bar of the message composition window. Then when the message is being queued, the translation is performed. No more editing is allowed on the message. All translators selected that have the Q4_transmission and Q4_completion flags will happen at this time.

When a plug-in is loaded, it registers in which of the above contexts each of its translators, attachers, and special tools wishes to be called in. For example a plug-in which does digital signing may have two translators in it, one to add a signature to an outgoing message, and one to verify a signature on an incoming message. The translator which adds the signature may register to be queued and called on transmission, and the signature verification translator may register to be called on-display.

The EMS API makes heavy use of the MIME standard for describing and representing the data type of an e-mail message and its sub-parts. The design of the API and the SDK is intended to make it possible to implement plug-ins without an in-depth understanding of MIME and without having to implement large parts of the MIME standard in the plug-ins.

Translators may operate on the whole message or only on any sub-part of the message. Eudora performs a full traversal of the MIME structure of the message and calls translators on parts and sub-parts as they wish to be called. This will allow plug-ins to work on individual parts of a multipart message without having to implement any MIME parsing.

The data type of a translator's input and output data is labeled using MIME. For example, the MIME typing might include the text format and character set, the type of compression, or the type of graphic image (e.g., GIF or JPEG).

Translators can create and access their own data files or make use of such files created by other applications. They may also access and modify data that is shared with a companion application.

In the case where the incoming message has no MIME structure, the message is transformed into a valid MIME structure with type text/plain. This is discussed in more detail within section 5.1.

### *1.2. Attachers*

When a message is being composed in Eudora, Attachers can be selected from a menu item. The plug-in will return file(s) that will be attached to the message.

### *1.3. Special Tools*

Special Tools will be available for selection on the menu at any time. This simply provides a hook for other utilities to be hooked into Eudora. Anything can be done here, launching another application, calling a script, etc.

# 2. Plug-ins, Translators, Attachers, Special Tools

Most individual translations that are a candidate for implementation via the EMS API come in pairs or groups. Examples are compression and decompression, Spanish to English and English to Spanish, digital signing and authenticating, and certificate management. Other plug-ins may implement attaching several kinds of items, or use several special tools, or some combination. An implementation of a group usually will have a lot of code in common and is most easily installed and configured by the user as a single entity. Thus, plug-ins are implemented as a collection of *translators, attachers and special tools.*

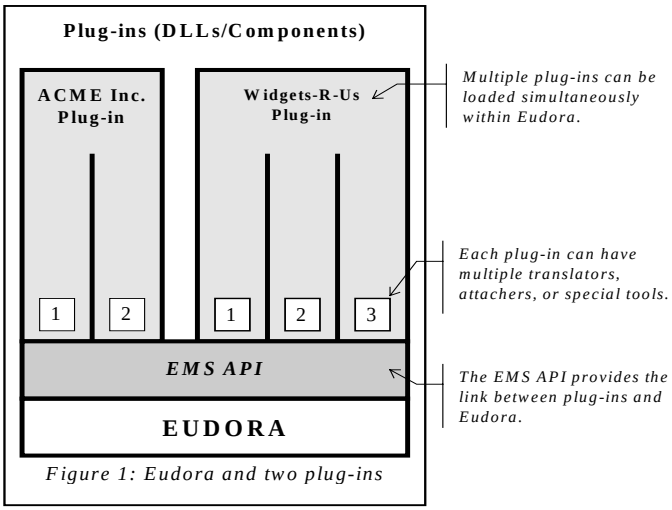

*Figure 1: Eudora and two plug-ins*

An individual translator in a plug-in performs one specific transformation on a translatable object. For example it authenticates the object, or converts a graphic from JPEG to GIF format. A plug-in is a collection of related translators. Plug-ins are implemented as a DLL for Windows and as a Component on the Macintosh.

### 2.1. Plug-in entry points

Each plug-in has a set of entry points or functions that are called by Eudora:

| | |
|---|---|
| `ems_plugin_version` | Is called first to get the API version number the plug-in uses and thereby the calling conventions for the other functions. `(Required)` |
| `ems_plugin_init` | Is always called second and only once as the plug-in is loaded during Eudora startup. `(Required)` |
| `ems_plugin_finish` | Called when Eudora exits. `(Required)` |
| `ems_free` | (Windows only) Called by Eudora to free data structures passed from the plug-in to Eudora. `(Optional)` |
| `ems_plugin_config` | Used to configure user-defined settings, called when the "Settings…" button is clicked while the plug-in is highlighted. `(Optional)` |

### 2.2. Translator entry points

| | |
|---|---|
| `ems_translator_info` | Supplies basic information about individual translators. Is called once for each translator on start up and at other times when specific items (like the icon) are needed for an individual translator. `(Optional*)` |
| `ems_can_translate` | Called to check whether a translation can be performed on a particular item, before the actual translation is attempted. |
| `ems_translate_file` | Called to actually perform the translation. |
| `ems_queued_properties` | For Q4-transmission translators, this allows user-defined properties to be set on a per-message basis. This function is called when the user clicks the translator icon while composing a message. `(Optional)` |

### 2.3. Attacher entry points

| | |
|---|---|
| `ems_attacher_info` | Called once on startup. This will add items to the Message-> Attachments sub-menu `(Optional*)` |
| `ems_attacher_hook` | When the Message->Attachments sub-menu item is called, this hook will be called to allow a file to be attached to a message. |

### 2.4. Special Tools entry points

| | |
|---|---|
| `ems_special_info` | Called once on startup. This will add items to the Special -> sub menu. `(Optional*)` |
| `ems_special_hook` | When the Special-> sub menu item is called. |

Some of these functions are optional, but every translator must supply a minimal set of these functions. The minimal set includes `ems_plugin_version`, `ems_plugin_init`, `ems_plugin_finish`. Except `ems_plugin_init`, `ems_plugin_finish`, and `ems_plugin_config` all of these functions take an argument which specifies which of the translators, attachers or special tools in the plug-in is being called. For example, if a plug-in was loaded that performs compression/decompression and Eudora wanted to call the data compression translator, it would call `ems_translate_file` with the ID of the compression translator. If it wanted to perform decompression it would also call `ems_translate_file`, but it would pass the ID of the decompression translator instead.

* At least one of `ems_translator_info, ems_attacher_info or ems_special_info` must be defined. In other words, this plug-in must have some actions associated with it.

### 2.5. Linking, loading and IDs

For Windows, a plug-in is implemented as a DLL. The above entry points are implemented as a set of functions in the DLL. A standard C calling convention is used, and the DLL is located by searching a specific set of directories (see section 8). The actual implementation may be in C, C++ or other, as long as the standard C calling convention is followed.

On the Macintosh, the EMS API makes use of the Component Manager to load and link the plug-in into Eudora. The calling convention thus conforms with what the Component Manager specifies. It is basically the stack-based Pascal calling convention. The details involved in implementing this can be skimmed over by using glue code supplied in the SDK. Plug-ins can be written in any language as long as the calling

convention is adhered to. Plug-ins may also be implemented from code fragments or shared libraries with some small amount of glue code. Exact details of what is needed to build a component are given in section six.

On the Macintosh it is also possible to statically link a plug-in with a test driver, the source for which is included in the SDK. It may be easier to debug plug-ins with the test driver since some of the Macintosh tools don't work as well on components.

Each plug-in must have a distinct ID number. To ensure these ID numbers are unique they are allocated by QUALCOMM. To obtain a unique ID, send a blank message to <emsapi-ids@qualcomm.com>. A list of several IDs will be returned by an auto-responder. The auto-responder doesn't actually track IDs by individuals or organization, it just returns monotonically increasing integers, so it's OK to request a second or third set if needed.

### 2.6. Stored State and access to other files

Plug-ins may permanently store configuration and other information as needed. Eudora provides no mechanism for this, but does suggest the name of a directory so plug-in configuration can track Eudora settings for users with multiple settings files. Basically, plug-ins should store state like any other application using a Preferences file or a `.INI` file. Shared configurations can be dealt with on a case-by-case basis depending on what is appropriate for the plug-in.

Plug-ins may also freely access other data and files and may share data with other applications. An example of this might be a set of dictionaries for language translation. Translators may also make accesses across the network. An example of this might be to access to directory service to get certificates.

### 2.7. Version numbering

There are version numbers for three things related to the EMS API. As Eudora changes it will have different version numbers. However every version of Eudora will not result in a change in the API definition so the API has its own version number. It is a single integer. It is also possible that Eudora will support multiple API versions for backward compatibility. The third version number is associated with the SDK. It may change independent of the Eudora version number. Both the Eudora version and the SDK version will change when the API version number changes. The current status is:

| | |
|---|---|
| **API:** | Current version is 4 |
| **Eudora:** | Macintosh versions 4.0 and higher support API **version 1, version 3, version 4** |
| | Windows versions 4.0 and higher support API **version 2, version 3, version 4** |
| **SDK:** | We provide an SDK for both the Macintosh and Windows on our web site at <http://www.eudora.com/developers/emsapi/> |

# 3. Translator Object Types and Formats

This section discusses the scheme used to describe the types and data formats of the input and output data that is actually translated. Most of the discussion centers on MIME, the Internet standard for encoding, structuring and typing data in Internet email.

The rest of the section is related to MIME. The translators use MIME in two ways. The first use of MIME is used to describe the type of the input and output data for a translator. All objects that are operated on via the EMS API have a MIME type. A translator usually determines what messages and message entities to operate on by the MIME. A translator must always specify the MIME type of its output when it returns the result to Eudora. These MIME types are passed to and from Eudora as parameters in the API entry point functions. Examples of types are `text/plain` for plain text, `image/gif` for a GIF image, and `multipart/signed` for an RFC-1847-style signed message. This pairing is referred to as the MIME type (e.g. "`text`") and the MIME subtype (e.g. "`plain`") when passed across the API.

The second use of MIME is for the format of the actual data. This is the data that is passed across the API by referencing a file name. The translated data can be in one of two basic formats, the native local format (e.g., plain text in the Macintosh character set or an unencoded GIF image), or in full MIME format (e.g., with MIME headers, canonicalization, and transfer encoding). It is expected that most translators need only operate on data in the local format, and thus do not need to do any MIME processing assigning and checking the MIME types as described above.

Plug-ins that operate on multipart MIME entities are the ones that will need to have their input and output data in MIME format. That is, the API uses standard MIME format to represent multipart MIME entities. One example of a plug-in that will require MIME format data is one that implements RFC-1847-style signed messages, since that format uses a two-part entity. One part is the signed data, and a second part is the signature. Another example is a plug-in that wishes to compress (or otherwise process) the full outgoing message including attachments.

## *3.1. Translated object types*

As mentioned above, each entity operated on by plug-ins is described by a MIME type, and this type is passed across the API in parameters to the entry point functions. (The term entity is used to refer to a message or a sub-part of a MIME message) The types are used by the translators to determine whether they should run on some data or not.

The EMS API defines a C data structure for passing MIME type information across the API to describe the data object being operated on. Source code for managing the data structure is available in the SDK.

When performing a translation, the plug-in will check the MIME type of the input data. This is usually the main criteria for the translator to decide whether or not it will perform the translation. The type is passed in by Eudora, so the translator doesn't actually have to examine the data to be translated. When the translation is complete, the translator must return the MIME type of the result to Eudora. Except for translators invoked in the on-request context, the MIME types for the input and output must be different (even if just by a MIME parameter) to avoid circular translations.

## *3.2. Translated object data formats*

This document has referred to the term *MIME entity*. This term comes from the MIME standard. In the simplest case a MIME entity is just an email message. The MIME standard assumes that a message with no MIME headers at all is a simple MIME entity of type `text/plain` with no transfer encoding or other MIME features. A multipart MIME message is also considered a MIME entity, as are each of its sub-parts.

If a message has nested multiparts, then each multipart is also a MIME entity. Basically a nested multipart MIME message can be viewed as having a tree structure, and every node in the tree (leaf or branch) is considered a MIME entity.

Plug-ins have the ability in certain contexts to translate any MIME entity in the structure of the message into a completely different MIME entity. A leaf node could be translated so that it is a multi-level, nested multipart entity. A message that has deeply nested MIME structure can be translated into a single text part.

It is expected that most translations will work on simple leaf MIME entities, those that do not have a top level type of `multipart`. In certain contexts, Eudora performs the traversal of the nested MIME structure and makes the data in the leaves available for translation so the plug-in author doesn't have to perform the traversal.

As is described in more detail later, each translator may be offered each MIME entity in the MIME structure to translate. It usually decides based on the MIME type whether or not it wishes to translate the entity. If the entity being translated is a multipart entity, then the data must be in MIME format. If it decides to translate the entity, the data is delivered in one of two formats as described in the next section.

### 3.3. Translated object data formats - the local non-MIME format

As mentioned earlier, data for a translator can be in one of two formats, one of which is the local or native non-MIME format. The local format is just the plain data as it normally is for the particular platform. Examples are Macintosh text (in Macintosh character set with CR line endings), DOS text, a JPEG file, or a Word document. Data in MIME format has additional headers and encoding as described below.

The actual format for each MIME entity is described by the standard or description for that MIME type (e.g., an image/gif entity will be described by the MIME standard for that type, which most likely references the standard for GIF images). Text formats however pose an unusual problem because they vary significantly between the Macintosh, Windows, etc. and there are no MIME documents describing local text formats. To solve this problem the translation API defines a type tag for the local text format for each platform.

On the Macintosh, the MIME type for text in the local format is `application/x-mac-text`. `Application/x-mac-text` has CR as the end of line and is in the Macintosh character set. The MIME type returned by an on-request text translator should be the same `application/x-mac-text.`

For Windows, text in the local format is of type `text/plain`, is in the ISO-8859-1 character set and has lines ending in CRLF. Similarly, the text returned by a translator should be in the same format and the MIME type should be `text/plain`.

At present, enriched text is removed before translation in the on-request context, but not other contexts.

The above is perhaps a complicated way of saying that on the Macintosh a simple text translator should accept and generate data of type `application/x-mac-text` and it can operate on data in standard Macintosh formats. Similarly for Windows it should accept and generate data of type `text/plain`.

### 3.4. Translated object formats - the MIME Canonical Format

MIME formatted data for translation is provided in all the translation contexts, except the on-request context, where the data is limited to text. When MIME formatted data is provided, Eudora supplies the data as follows:

- Converts the base data objects to their canonical format as defined by its MIME type and subtype. The most common canonicalization is to convert text so the line endings are CRLF and the character set to a standard one like ISO-8859-1.

- It applies content transfer encoding so the result is 7-bit clean limited line length data. This is done using Eudora's usual algorithm for determining which transfer encoding is best. Eudora uses quoted-printable transfer encoding for text data and base-64 for non-text data. Whether the data is text or not is determined by the MIME type mapping settings in Eudora.

- It assembles the MIME entity with the appropriate MIME headers. These consist of the `MIME-Version`, `Content-type`, and `Content-transfer-encoding` headers with appropriate parameters, message part boundaries, etc.

Translators that return full MIME should return similar entities. The `MIME-version` header should always be included with one exception. The MIME version header should not be output by a translator for translations on outgoing messages on the Macintosh. Macintosh Eudora always assumes MIME version 1 and generates the header for its outgoing messages. If the `Content-type` is omitted `text/plain` will be assumed, and if the `Content-transfer-encoding` is omitted, `7bit` will be assumed. Note that the entities Eudora supplies will always be encoded for 7bit transport, however the translator can return the entity with any standard transfer encoding as long as it is tagged correctly. Other MIME-related `Content-*` headers can be included.

Below is an example of text in the MIME format. The lines would end with CRLF and the data would be in this format no matter if the translation is being done on the Macintosh or Windows. If it were not in MIME format it would not have the extra header, nor the quoted printable transfer encoding, and the character set might not be ISO-8859-1.

```
Content-type: text/plain; charset=iso-8859-1
Content-transfer-encoding: quoted-printable

This is the message text and this =e1 is an a with an accent.
```

The API uses the tag `text/plain` for the local format for Windows because the character set and end of line character are the same as the Internet standard. The above entity in Windows local format would be as follows and has no header or transfer encoding.

```
This is the message text and this is á an a with an accent.
```

# 4. Display in the User Interface

Plug-ins and translators are displayed in the user interface in several places. On the Mac all plug-ins are shown in the About Extended Message Services dialog box found under the apple menu. On Windows the Message Plug-in Settings dialog is accessible under the Special menu.

Translators that can operate in the ON_REQUEST context are displayed as menu items. They are enabled for received messages, messages under composition, and most any editable text field found throughout Eudora. When invoked they are performed immediately on the current text field (eg. a composition message, a received message, etc.). These menu items are only active when the user's focus is in an editable text field and should not be used a general hook for adding menus to Eudora.

Translations that can be operated in the Q4-transmission and Q4-completion context are displayed as either checkable icons (Macintosh) or as state icons(Windows) in the toolbar of the message composition window. While the user is composing the message, they may be selected and deselected. Q4-completion icons will have a light gray outline around them in Windows.

The on-request translators may return a text message. If this is returned it will be displayed as part of the message.

Some translators operate without any user interface. These are translators that work in the on-arrival context. They process messages as they are down-loaded from the mail server.

*Attachers* appear in the Message -> Attach sub-menu as menu items with the description on the menu. They are always enabled when a new message is being composed, it simply attaches the returned files to the message. When there is no new message in front, the Windows version will create a new message, then call the attacher.

*Special Tools* appear in the Tool Menu on Windows and the Special Menu on Mac as menu items with the description on the menu. They are always enabled.

*ON_REQUEST*, *Attachers* and *Special Tool* Translators can appear on the main toolbar by setting the flag EMSF_TOOLBAR_PRESENCE which will automatically add them when Eudora starts up.

# 5. The Translation Process

A translator supplies two functions that are used in the translation process itself, `ems_can_translate` and `ems_translate_file`.

Translations may be performed in different *contexts*. These contexts are different events that happen to a message, such as its arrival, display, or transmission. The details for each are described below. A given translator can work in any number of these contexts. When a translator is called by Eudora the context it is being called in is specified by a parameter so it may behave differently in different contexts.

When Eudora processes a message for translation the function `ems_can_translate` is called for each potentially translatable MIME entity before actual translation is attempted. In some cases this is for the sake of efficiency since `ems_can_translate` is more efficient than the full translation function. The function `ems_can_translate` also has a special return code, namely `EMSR_NOT_NOW`, to delay further processing of a message to a later time. The main purpose of `EMSR_NOT_NOW` is for a translator to delay all further MIME parsing and translation. This may occur because the translator works on unparsed MIME entities. It may also wish to preempt translation in a non-interactive context so the translation can be performed later in a context where interaction with the user is allowed. *Note that on-display translators are **required** to return `EMSR_NOT_NOW` in the on-arrival context.*

The function `ems_translate_file` actually performs the translation. It is passed a large number of parameters, including the input MIME type, the location of the data to translate, the address of a progress reporting function, and the e-mail addresses on the message. Exact details are given in Section 8.

## *5.1. On-arrival*

The on-arrival context processes messages as they are down-loaded to Eudora from the mail server. That is, when Eudora is talking to the POP server. In general, translators in this context should not interact with the user or cause long delays (more than a few seconds) or they will disrupt the POP protocol session with the mail server. This context is useful for automatically processing incoming messages. It is also necessary to use this context so that translations can be performed in the on-display context.

The actual algorithm used by Eudora to call translators is integrated with Eudora's MIME parsing. It involves a pre-order traversal of the MIME structure of the message (intermediate nodes are processed before the leaves). As each MIME entity is visited the `ems_can_translate` function of each translator is called on it. If it returns `EMSR_CANT_TRANS`, the next translator is tried. The list of translators that are tried are the ones that indicate they work in the on-arrival context and are ordered by type as listed below. If `ems_can_translate` returns `EMSR_NOW`, the translation is immediately performed and the output of the translator replaces the MIME entity that was translated. After a translation is made, the entire process of checking each translator in the list at each node in the pre-order traversal is started over for that MIME entity. When a complete pass is made through all translators for an entity without performing any translation, the MIME parse of the entity is made and its sub-parts are processed. Since most messages are not multipart and most will not be translated, this usually amounts to a single pass through the potential translators.

If the `ems_can_translate` function returns `EMSR_NOT_NOW`, then all parsing stops and the MIME entity as it stands is written out for later processing. The entity is written to a file and a link to the file is placed in the original message. When the user clicks on the link, the translation process is resumed in the on-display context for the same translator.

In general, the order of the translations in the on-arrival context is driven by the MIME types in the received message. When there are ambiguities, the order is by type as follows:

```
EMST_CERT_MANAGEMENT                (first)
EMST_PREPROCESS
EMST_SIGNATURE
EMST_COALESCED
EMST_COMPRESSION
EMST_GRAPHIC_FORMAT
EMST_TEXT_FORMAT
EMST_LANGUAGE                       (last)
```

Translations in the on-arrival context should not interact with the user. If they need to interact with the user they should delay processing until the on-display context by returning `EMSR_NOT_NOW`. A translator may also vary the function it performs based on the context in which it is called. For example a signature verification translator called in the on-arrival context may find it useful to fail silently if it does not have the certificate needed for verification rather than interrupt the message down-load to prompt for a certificate.

Translators in this context must accept MIME and generate MIME. That is, the `EMSF_REQUIRES_MIME` and `EMSR_GENERATES_MIME` flags are ignored and Eudora treats the translator as if they were set. Thus these translators must be prepared to remove content transfer encoding, and parse and generate basic MIME structure.

In the case where the received message is not MIME (missing the “MIME-Version” header), Eudora will convert the message to MIME. This is done by adding the MIME-Version and Content-Type headers, using the type text/plain. This transformation happens before any translation is done to the message, which means your translator cannot distinguish between non-MIME messages which have been coerced into MIME and messages originally MIME.

### 5.2. On-display

Translations in the on-display context are performed when a user clicks on a translator icon that appears in a message body. The translator icon is put in the message body as a result of the `ems_can_translate` function called in the on-arrival context returning `EMSR_NOT_NOW`. When the user clicks on the icon, the parsing, recursion, and translation on the MIME structure that was begun in the on-arrival context is resumed. When the traversal is complete the resulting MIME entity is parsed and text parts are displayed to the user, in the message window. This includes icons for attachments that were part of the original message or attachments that were generated as part of the translation process. Attachments can also be removed as part of the translation process.

Important differences between this context and the on-arrival context are that `ems_can_translate` must never return `EMSR_NOT_NOW` and that translations may interact with the user. The on-display context has the same restriction as the on-arrival context that the input and the output must be MIME format.

*When Eudora messages are stored in MIME format, translations in this context may be performed automatically when the message is displayed - that is when the user clicks on the message index to display a particular message. There will be no need for the user to click on a translator icon in the message body.*

### 5.3. On-request

On-request translations are those that are performed on the currently displayed message. Translations in this context are usually the simplest to create.

Translators that work in this context are displayed in a menu item in a sub-menu of the Edit menu. When the user selects one, the translation is performed on the current message whether it is a received message or a message under composition. If a section of the message is selected, then only the selection is processed.

When complete, the translated data replaces the original data and the message is marked as changed. Translations in this context may be fully interactive. If there is no open current message or the user's focus is not in an editable text field, then the menu items for these translations are disabled.

Under Windows, Eudora determines which translators are placed in the Edit menu by calling `ems_can_translate` with the MIME type `text/plain, and text/html`. for each on-request translator. If the result is `EMSR_NOW`, the translator will be placed in the menu. On the Macintosh, all on-request translators are placed in the Edit menu, no MIME type checking is done.

Under Windows, an on-request translators may set the `EMSR_REQUIRES_MIME` and `EMSR_GENERATES_MIME` flags or not – they are ignored. Regardless of these flags the data type will be `text/plain` and the line endings will be CRLF and the translator should return the same to Eudora.

On the Macintosh, on-request translators can use the local format (described in section 3.3) if the `EMSR_REQUIRES_MIME` or `EMSR_GENERATES_MIME` flags are set.

If the message is text/html, it will first be offered to the translator as text/html. It can either accept it with `EMSR_NOW` or decline it with `EMSR_CANT_TRANSLATE`. If it is declined, Eudora will convert the message to text/plain and offer it that way. Either text/plain or text/html can be returned and Eudora will put it back in the message appropriately. This is not the case for other translation contexts.

### *5.4. Queue and call on transmission (Q4-transmission)*

Translators that work in this context are displayed in the toolbar of the composition windows and may be selected by the user. They are toggled on and off by clicking a button with the translator's icon on it (Macintosh) or by selecting the translation from a drop-down menu (Windows). The translation is actually performed later when the message is being transmitted to the mail server via SMTP. If a message under composition is saved and resumed later, the toggled state of all translators working in the Q4-transmission context will be retained.

Translation in this context must operate on the full MIME structure and must work on the whole message (must set `EMSF_GENERATES_MIME`, `EMSF_REQUIRES_MIME` and `EMSF_WHOLE_MESSAGE`). Translations are performed in the reverse of the order listed above for on-arrival translations. This ordering does prevent certain useful chains of translations from being performed (e.g., first a language translation, then a text format translation), but this disadvantage is out weighed by it being simpler to implement, and simpler for the user.

The `EMSF_WHOLE_MESSAGE` flag indicates a Q4-transmission translator wishes to operate on the whole message, thus it will not be offered the intermediate nodes for translation. ***NOTE**: This flag is required for all Q4-transmission translators.*

The `ems_can_translate` function for this context is called after the user clicks the Send/Queue button. This allows the translator to perform a quick check that the translation will be possible later when the message is transmitted. If `ems_can_translate` returns `EMSR_CANT_TRANS` and an error string, the string will be displayed to the user, and the message will not be sent or queued. The user has the option of toggling the translation off or adjusting the condition that caused the translation to fail.

It is possible for the user to queue an incompatible set of translations (e.g., the MIME type output by one translation is not acceptable input to the next).When this happens the user will receive an error and can then go back and deselect translations.

Translations in this context may be fully interactive.

### 5.5. Queue and call on completion (Q4-completion)

Translators that work in this context are displayed in the toolbar of the composition windows and may be selected by the user. They are toggled turned on and off by clicking a button with the translator's icon on it. The translation is actually performed when the Queue/Send button is clicked on. All Q4-transmission and Q4-completion translations will occur and the message will become a MIME message attached to the outgoing message. It will no longer be editable.

Translation in this context must operate on the full MIME structure and must work on the whole message (must set `EMSF_GENERATES_MIME`, `EMSF_REQUIRES_MIME` and `EMSF_WHOLE_MESSAGE`). Translations are performed in the reverse of the order listed above for on-arrival translations. This ordering does prevent certain useful chains of translations from being performed (e.g., first a language translation, then a text format translation), but this disadvantage is out weighed by it being simpler to implement, and simpler for the user.

The `ems_can_translate` function for this context is called after the user clicks the Send/Queue button. This allows the translator to perform a quick check that the translation will be possible later when the message is transmitted. If `ems_can_translate` returns `EMSR_CANT_TRANS` and an error string, the string will be displayed to the user, and the message will not be sent or queued. The user has the option of toggling the translation off or adjusting the condition that caused the translation for fail.

It is possible for the user to queue an incompatible set of translations (e.g., the MIME type output by one translation is not acceptable input to the next).When this happens the user will receive an error and can then go back and deselect translations.

# 6. Attacher Plug-ins

When a plug-in includes any attachers, there will be a menu item in the Message ➔ Attach sub-menu for each attacher. The total number of attachers in the plug-in is returned in `ems_plugin_init`. For each attacher, `ems_attacher_info` is called. When a user in composing a message, these items will be enabled. If an attacher menu item is selected, `ems_attacher_hook` is called, and the plug-in can provide a UI for selecting or creating file(s) or any special media types.

When Eudora sends an attachment, it determines the MIME type/subtype by either looking up the file extension in a MIME mapping table (Windows) or looking for specific resources inside Eudora (Macintosh); see Appendix B for more information. Eudora will handle all processing of making the attachment a MIME message so it can be sent out over the Internet.

To insure a specific MIME structure of the message, an attacher can create a file that is a fully-formatted MIME part, and identify it with the .MSG suffix on Windows, or the 'MiME' type with 'CSOm' creator on the Mac. These files must be **complete** MIME parts, all encoding, canonicalization, MIME headers, etc. must be present in the file. Eudora will not do any further processing on the file, it will simply put it on the wire as it's sending out the message. This allows the user to create any MIME format, including complex, nested multipart MIME structure. Having this control also allows the attacher to specify the disposition as inline versus attachment via the 'Content-Disposition' header [DISP].

# 7. Special Tools Plug-ins

When Special Tools exist within a plug-in, they will be placed on the Tools menu in Windows and on the Special Menu on the Mac. These items will always be enabled and available to the user. The number of Special Tools within a plug-in is returned in `ems_plugin_init`. For each Special Tool, `ems_special_info` is called. If a Special Tool item is selected, `ems_special_hook` is called, and the plug-in can do as it likes. Eudora will wait until the `ems_special_hook` function returns.

# 8. API Reference

This section describes in full detail the calling interface, constants and related data structures. These definitions are the same as found in the include `emsapi-mac.h` and `ems-win.h`. The basic data items and their semantics for the API do not vary between the Macintosh and Windows platforms, but the function declarations and data formats do vary. Having this variance between platforms makes the API simpler and less abstract for each platform, and also increases its efficiency. In the following sections both the Macintosh and Windows declarations are shown.

For both the Mac and Windows platforms, header files, skeleton source code, and samples are part of the SDK. In particular, this should help with some of the complexity in working with the Macintosh Component manager. The author should be able to create a plug-in by creating the necessary C functions and some associated resources.

## *8.1. Constants*

The first three letters, `EMS`, identify EMS API-related constants. The fourth letter groups related constants. All constants should be stored as a `long` (32 bits). The constants are identical on all platforms.

Return codes report the general success or failure of a translation and are not intended to express all possible results of a translation. Plug-ins can also pass text messages back to Eudora to be displayed to the user.

```
/* ----- Return codes --- store as a long ---------------------- */
#define EMSR_OK              (0L)     /* The translation operation succeeded /
#define EMSR_UNKNOWN_FAIL    (1L)     /* Failed for unspecified reason */
#define EMSR_CANT_TRANS      (2L)     /* Don't know how to translate this */
#define EMSR_INVALID_TRANS   (3L)     /* The translator ID given was invalid /
#define EMSR_NO_ENTRY        (4L)     /* The value requested doesn't exist */
#define EMSR_NO_INPUT_FILE   (5L)     /* Couldn't find input file */
#define EMSR_CANT_CREATE     (6L)     /* Couldn't create the output file */
#define EMSR_TRANS_FAILED    (7L)     /* The translation failed. */
#define EMSR_INVALID         (8L)     /* Invalid argument(s) given */
#define EMSR_NOT_NOW         (9L)     /* Translation can be done not in current
                                         context */
#define EMSR_NOW            (10L)     /* Indicates translation can be performed
                                         right away */
#define EMSR_ABORTED        (11L)     /* Translation was aborted by user */
#define EMSR_DATA_UNCHANGED (12L)     /* Trans OK, data was not changed */
```

Every translator (not attachers, nor special tools) must be one of the following types. The type is used to determine the ordering of translations in certain contexts when ambiguities arise (see the previous section on *The Translation Process)*. When, in a particular plug-in, translators of type `EMST_SIGNATURE` and `EMST_PREPROCESS` are selected together in the `EMSF_Q4_TRANSMISSION` or `EMSF_Q4_COMPLETION` context, and a translator of type `EMST_COALESCED` is available it will be called instead of the two translators. Translators of type `EMST_COALESCED` should not supply an icon if it is desired that they not be displayed and selectable on the composition window. Basically the translation types are used for ordering and grouping the translations and for nothing else.

```
/* ----- Translator types --- store as a long ---------------------------- */
#define EMST_NO_TYPE          (-1L)
#define EMST_LANGUAGE         (0x10L)
#define EMST_TEXT_FORMAT      (0x20L)
#define EMST_GRAPHIC_FORMAT   (0x30L)
#define EMST_COMPRESSION      (0x40L)
#define EMST_COALESCED        (0x50L)
#define EMST_SIGNATURE        (0x60L)
#define EMST_PREPROCESS       (0x70L)
#define EMST_CERT_MANAGEMENT  (0x80L)
```

The following flags specify critical information about a translator. They specify which context it may operate in, whether or not it can be called on the whole message or not, and the format of the input and

output data. Eudora uses these flags to decide when to call the translator, and how to format and process the input and output data from the translator.

```
/* ----- Translator info flags and contexts --- store as a long ---------- */
/* Used both as bit flags and as constants */
#define EMSF_ON_ARRIVAL      (0x0001L) /* Call on message arrivial */
#define EMSF_ON_DISPLAY      (0x0002L) /* Call when user views message */
#define EMSF_ON_REQUEST      (0x0004L) /* Call when selected from menu */
#define EMSF_Q4_COMPLETION   (0x0008L) /* Queue and call on complete
                                          composition of a message */
#define EMSF_Q4_TRANSMISSION (0x0010L) /* Queue and call on transmission
                                          of a message */
#define EMSF_WHOLE_MESSAGE   (0x0200L) /* Works on the whole message even if
                                          it has sub-parts. (e.g. signature) */
#define EMSF_REQUIRES_MIME   (0x0400L) /* Items presented for translation
                                          should be MIME entities with
                                          canonical end of line representation,
                                          proper transfer encoding
                                          and headers */
#define EMSF_GENERATES_MIME  (0x0800L) /* Data produced will be MIME format */
#define EMSF_ALL_HEADERS     (0x1000L) /* All headers in & out of trans when
                                          MIME format is used */
#define EMSF_BASIC_HEADERS   (0x2000L) /* Just the basic headers */

#define EMSF_DEFAULT_Q_ON    (0x4000L) /* Causes queued translation to be on
                                          for a new message by default */
 #define EMSF_TOOLBAR_PRESENCE(0x8000L) /* Automattically appear on the Toolbar when
                                           Eudora starts up*/
#define EMSF_ALL_TEXT         (0x10000L)/* ON_REQUEST WANTS WHOLE MESSAGE */

/* all other flag bits in the long are RESERVED and may not be used */
```

The final following constants define the API version number, the component type used on the Macintosh, and the `out_codes` that should be returned from `ems_translate` when called on a translator of type `EMST_SIGNATURE`. The component type goes in the `thng` resource of the component.

```
/* ----- The version of the API defined by this include file ------------- */
#define EMS_VERSION          (4)       /* Used in plug-in init */
#define EMS_COMPONENT        'EuTL'    /* Macintosh component type */
```

## 8.2. Macintosh data structures

```
/* ----- MIME Params --------------------------- */
typedef struct emsMIMEparamS *emsMIMEParamP, **emsMIMEparamH;
typedef struct emsMIMEparamS {
      long               size;
      Str63              name;          /* MIME parameter name */
      Handle             value;         /* handle size determines string length */
    emsMIMEparamH        next;          /* Handle for next param in list */
} emsMIMEparam;

/* ----- MIME Data ----------------------------- */
typedef struct emsMIMEtypeS *emsMIMEtypeP, **emsMIMEtypeH;
typedef struct emsMIMEtypeS {
      long               size;
      Str63              mimeVersion;   /* MIME-Version: header */
      Str63              mimeType;      /* Top-level MIME type: text,message...*/
      Str63              subType;       /* sub-type */
      emsMIMEparamH      params;        /* Handle to first parameter in list */
      Str63              contentDisp;   /* Content-Disposition */
      emsMIMEparamH      contentParams;  /* Handle to first parameter in list */
} emsMIMEtype;


/* ----- User Address ------------------------ */
typedef struct emsAddressS *emsAddressP, **emsAddressH;
typedef struct emsAddressS {
      long               size;          /* Size of this data structure */
      StringHandle       address;       /* Optional directory for config file */
      StringHandle       realname;      /* Users full name from Eudora config */
      emsAddressH        next;          /* Linked list of addresses */
} emsAddress;

/* ----- Header Data --------------------------- */
typedef struct emsHeaderDataS *emsHeaderDataP, **emsHeaderDataH;
```

```
typedef struct emsHeaderDataS {
      long              size;          /* Size of this data structure */
      emsAddressH       to;            /* To Header */
      emsAddressH       from;          /* From Header */
      StringPtr         *subject;      /* Subject Header */
      emsAddressH       cc;            /* cc Header */
      emsAddressH       bcc;           /* bcc Header */
      Handle            rawHeaders;    /* The 822 headers */
} emsHeaderData;


/* ----- How Eudora is configured ------------- */
typedef struct emsMailConfigS *emsMailConfigP, **emsMailConfigH;
typedef struct emsMailConfigS {
      long              size;          /* Size of this data structure */
      FSSpec            configDir;     /* Optional directory for config file */
      emsAddress        userAddr;      /* Current users address */
} emsMailConfig;


/* ----- Plugin Info -------------------------- */
typedef struct emsPluginInfoS *emsPluginInfoP, **emsPluginInfoH;
typedef struct emsPluginInfoS {
      long              size;          /* Size of this data structure */
      long              id;            /* Place to return unique plugin id */
      long              numTrans;      /* Place to return num of translators */
      long              numAttachers;  /* Place to return num of attach hooks */
      long              numSpecials;   /* Place to return num of special hooks */
      StringHandle      desc;          /* Return for string description of plugin */
      Handle            icon;          /* Return for plugin icon data */
} emsPluginInfo;


/* ----- Translator Info ---------------------- */
typedef struct emsTranslatorS *emsTranslatorP, **emsTranslatorH;
typedef struct emsTranslatorS {
      long              size;          /* Size of this data structure */
      long              id;            /* ID of translator to get info for */
      long              type;          /* translator type, e.g., EMST_xxx */
      unsigned long     flags;         /* translator flags */
      StringHandle      desc;          /* translator string description */
      Handle            icon;          /* translator icon data */
      StringHandle      properties;    /* Properties for queued translations */
} emsTranslator;


/* ----- Menu Item Info ---------------------- */
typedef struct emsMenuS *emsMenuP, **emsMenuH;
typedef struct emsMenuS {
      long              size;          /* Size of this data structure */
      long              id;            /* ID of menu item to get info for */
      StringHandle      desc;          /* translator string description */
      Handle            icon;          /* translator icon data */
      long              flags;         /* any special flags*/
} emsMenu;


/* ----- Translation Data --------------------- */
typedef struct emsDataFileS *emsDataFileP, **emsDataFileH;
typedef struct emsDataFileS {
      long              size;          /* Size of this data structure */
      long              context;
      emsMIMEtypeH      mimeInfo;      /* MIME type of data to check */
      emsHeaderDataP     header;
      FSSpec            file;          /* The input file name */
} emsDataFile;


/* ----- Resulting Status Data ---------------- */
typedef struct emsResultStatusS *emsResultStatusP, **emsResultStatusH;
typedef struct emsResultStatusS {
      long              size;   /* Size of this data structure */
      StringHandle      desc;   /* Returned string for display with the result */
      StringHandle      error;  /* Place to return string with error message */
      long              code;   /* Return for translator-specific result code */
} emsResultStatus;


/* ----- Progress Data ----------------------------- */
typedef struct emsProgressDataS *emsProgressDataP, **emsProgressDataH;
```

```
typedef struct emsProgressDataS {
      long               size;          /* Size of this data structure */
      long               value;         /* Range of Progress, percent complete */
      StringPtr          message;       /* Progress Message */
} emsProgressData;
```

On the Macintosh, strings passed from a translator to Eudora (such as descriptions, error messages and email addresses) are Pascal strings. Eudora will pass a pointer to the location where the `Handle` to the string should be returned. The translator must allocate this `Handle` with `NewHandle()` so that Eudora can free it with `DisposeHandle()`.

File path names are not used. Instead Eudora passes a pointer to an `FSSpec` on the stack. (Translators never return file names to Eudora).

The structures representing a MIME type are also `Handles` allocated with `NewHandle()`. Limited-length Pascal strings are used for all components of the MIME type, except for parameter values. The parameter value is a `Handle` to a string the length of which is determined by the size of the `Handle`. The parameter value is not a Pascal string because its length can potentially exceed that of a Pascal string. It is also not NULL-terminated as the length comes from the handle size.

When Eudora passes a pointer to a location in which it expects data to be returned by a translator, it may pass NULL. Translators must check that the pointer to the location is not NULL before placing a value in it.

### 8.3. Windows data structures

```
/* ----- MIME Params --------------------------- */
typedef struct emsMIMEparamS FAR*emsMIMEParamP;
typedef struct emsMIMEparamS {
      long               size;
      LPSTR              name;          /* Mime parameter name (e.g., charset) */
      LPSTR              value;         /* param value (e.g. us-ascii) */
      emsMIMEParamP      next;          /* Linked list of parameters */
} emsMIMEparam;


/* ----- MIME Info ----------------------------- */
typedef struct emsMIMEtypeS FAR*emsMIMEtypeP;
typedef struct emsMIMEtypeS {
      long               size;
      LPSTR              version;       /* The MIME-Version header */
      LPSTR              type;          /* Top-level MIME type */
      LPSTR              subType;       /* sub-type */
      emsMIMEParamP      params;        /* MIME parameter list */
      LPSTR              contentDisp;   /* Content-Disposition */
      emsMIMEParamP      contentParams; /* Handle to first parameter in list */
} emsMIMEtype;


/* ----- User Address -------------------------- */
typedef struct emsAddressS FAR*emsAddressP;
typedef struct emsAddressS {
      long               size;          /* Size of this data structure */
      LPSTR              address;       /* Optional directory for config file */
      LPSTR              realname;      /* Users full name from Eudora config */
      emsAddressP        next;          /* Linked list of addresses */
} emsAddress;


/* ----- Header Data --------------------------- */
typedef struct emsHeaderDataS FAR*emsHeaderDataP;
typedef struct emsHeaderDataS {
      long               size;          /* Size of this data structure */
      emsAddressP        to;            /* To Header */
      emsAddressP        from;          /* From Header */
      LPSTR              subject;       /* Subject Header */
      emsAddressP        cc;            /* cc Header */
      emsAddressP        bcc;           /* bcc Header */
      LPSTR              rawHeaders;    /* The 822 headers */
} emsHeaderData;


/* ----- How Eudora is configured -------------- */
typedef struct emsMailConfigS FAR*emsMailConfigP;
typedef struct emsMailConfigS {
      long               size;          /* Size of this data structure */
      HWND               FAR*eudoraWnd; /* Eudora's main window */
      LPSTR              configDir;     /* Optional directory for config file */
      emsAddress         userAddr;      /* Users full name from Eudora config */
} emsMailConfig;


/* ----- Plugin Info --------------------------- */
typedef struct emsPluginInfoS FAR*emsPluginInfoP;
typedef struct emsPluginInfoS {
      long        size;                 /* Size of this data structure */
      long        numTrans;             /* Place to return num of translators */
      long        numAttachers;         /* Place to return num of attach hooks */
      long        numSpecials;          /* Place to return num of special hooks */
      LPSTR       desc;                 /* Return for string description of plugin */
      long        id;                   /* Place to return unique plugin id */
      HICON       FAR*icon;             /* Return for plugin icon data */
} emsPluginInfo;


/* ----- Translator Info ----------------------- */
typedef struct emsTranslatorS FAR*emsTranslatorP;
typedef struct emsTranslatorS {
      long               size;          /* Size of this data structure */
      long               id;            /* ID of translator to get info for */
      long               type;          /* translator type, e.g., EMST_xxx */
      ULONG              flags;         /* translator flags */
      LPSTR              desc;          /* translator string description */
      HICON              FAR*icon;      /* translator icon data */
```

```
      LPSTR              properties;    /* Properties for queued translations */
} emsTranslator;


/* ----- Menu Item Info --------------------------- */
typedef struct emsMenuS FAR*emsMenuP;
typedef struct emsMenuS {
      long               size;          /* Size of this data structure */
      long               id;            /* ID of translator to get info for */
      LPSTR              desc;          /* translator string description */
      HICON              FAR*icon;      /* Return for plugin icon data */
      long               flags;         /* any special flags*/
} emsMenu;

/* ----- Translation Data ---------------------------- */
typedef struct emsDataFileS FAR*emsDataFileP;
typedef struct emsDataFileS {
      long               size;          /* Size of this data structure */
      long               context;
      emsMIMEtypeP       info;          /* MIME type of data to check */
      emsHeaderDataP      header;
      LPSTR              fileName;      /* The input file name */
} emsDataFile;


/* ----- Resulting Status Data ---------------------------- */
typedef struct emsResultStatusS FAR*emsResultStatusP;
typedef struct emsResultStatusS {
      long               size;   /* Size of this data structure */
      LPSTR              desc;   /* Returned string for display with the result */
      LPSTR              error;  /* Place to return string with error message */
      long               code;   /* Return for translator-specific result code */
} emsResultStatus;


/* ----- Progress Data ---------------------------- */
typedef struct emsProgressDataS FAR* emsProgressDataP;
typedef struct emsProgressDataS {
      long               size;          /* Size of this data structure */
      long               value;         /* Range of Progress, percent complete */
      LPSTR              message;       /* Progress Message */
} emsProgressData;
```

For Windows, ASCII strings for descriptions, error messages, file names, addresses and components of the MIME type structure are all NULL-terminated strings. They may be allocated any way the plug-in author wishes and is referred to as the plug-in's internal allocator. Eudora will call `ems_free` as supplied by the plug-in to free the storage when it is finished with the data.

The icons returned by `ems_plugin_init` for the whole plug-in should be a 32x32 `HICON`. The icons for the individual translators should be a 16x16 `HICON` (creating the 16x16 `HICON` may involve creating a `HICON` and deleting the 32x32 part). All the icons should be allocated with the plug-ins internal allocator so Eudora can free them by calling `ems_free`.

When Eudora passes a pointer to a location in which it expects data to be returned by a translator, it may pass NULL. Translators must check that the pointer is not NULL before placing a value in it.

### *8.4. Building Macintosh components*

As mentioned previously, plug-ins on the Macintosh are implemented as Components. Components are used, rather than other mechanisms such as Code Fragments, because they work on all Macintosh hardware from the 68000 to the PowerPC, and on MacOS system 7.0 through current versions. It is also expected they will be supported in future versions of MacOS. Though creating a component can be complicated, the SDK provides most of the needed glue source code, and the job should be easier.

In general the plug-in author needs to implement a minimal set of the entry point functions. When the Component is built the `thng` resource of the component must have type '`euTL`'. The version number

specified in the thng resource must be a valid translation API version number. The upper 16 bits can be set to the value of the constant EMS_VERSION from the API include files. The sub-type resource is not used, but it must be unique or the translator will not be loaded by the Component Manager. There is currently no registry for sub-types to guarantee their being unique, but this not expected to be a problem. The author should make one up of their own. It must not be all lower case letters as those are reserved by Apple. Other fields of the component resource such as flags, icon, and descriptions are ignored.

The SDK includes two files for building a plug-in. The first, emsapi-mac.h, includes the constants and data structures listed here. It includes prototypes for the eight functions that are needed. For building the translator as a component, the file ems-component.c can be used as the component main. It includes the necessary component manager glue to accept the standard component manager calls as well as the API calls. When it receives the API calls, it sets up the calling stack frame and then calls the functions which are proto-typed in emsapi-mac.h. Thus ems-component.c should be compiled as a normal C file and linked into the component.

In order to compile ems-component.c, the template file usertrans.h must be modified for the plug-in being authored. A sample is included. It contains two sections. One is the definition of the structure tlUserGlobals. This is a structure that is passed as the first argument for all the API calls. The translator can define data it wants to be carried between calls to the API and store it here. This structure is automatically allocated and managed by the component manager glue in ems-component.c. Also in usertrans.h are C pre-processor definitions for eight constants that indicate whether an API call is implemented by the particular plug-in. Each constant should be defined to either true or false.

Eudora looks in a pre-defined set of directories for the Components that are EMS API plug-ins. This is done at start-up time. Each plug-in discovered is loaded and becomes active. The plug-ins must have a thng resource as described above or they will not be loaded. For the Macintosh, the paths are:

```
the folder the Eudora application is in
the sub folder Eudora Stuff of the folder the application is in
the extensions folder in the active system folder
```

Note that the Eudora folder (where Eudora stores mailboxes and related files, but not the application file) is **not** searched for plug-ins!

### 8.5. Building Windows DLLs

Building a translation DLL is straightforward because all that is needed is a DLL that implements a minimal subset of the API entry point functions using the standard "C" calling convention.

Eudora looks in a pre-defined set of directories for Windows DLLs that are EMS API plug-ins. This is done at start-up time. Each plug-in discovered is loaded and becomes active. For Windows the directories are:

```
The sub-directory "plugins" of the directory the Eudora .exe file is in
The sub-directory "plugins" of the mail directory
```

The fact that a particular DLL is an EMS API DLL is determined by checking that it implements the ems_plugin_version, ems_plugin_init and one of ems_translator_info, ems_attacher_info or ems_special_info functions.

When creating icons, Eudora supports a 256 color palette which can be found in the file safety.bmp. Most paint programs, including Paint Shop Pro and Adobe Photoshop, can use this file to extract the palette. The reserve entries that shouldn't be used are indexes 11,12,13, and 14 (first color is index 0). An image may use a different palette, but its colors will be mapped into Eudora's regardless if shown in Eudora with the screen mode set to 256 colors (8 bit color).

### 8.6. Efficiency considerations

Most of the functions in a plug-in, except the actual translation, can usually be implemented with a very small amount of code. These functions are also called much more frequently than the actual translation functions. Thus in some cases it may be advantageous to implement a translator in two parts, the smaller part which is loaded in memory all the time, and the larger part which is only loaded when translations are to be performed.

On the Macintosh, this second part can be another component, a shared library or a code fragment. Nothing about the API precludes any of these, and it is up to the translator author to decide which is to be used based on which platforms are to be supported.

A similar strategy may be adopted with Windows where the bulk of the translation function is implemented as a second DLL that is loaded only when a translation is being performed.

### *8.7. Get the API version number that this plug-in implements*

Macintosh:

```
pascal long ems_plugin_version(
short *api_version          /* Out: Place to return api version */
);
```

Windows:

```
extern "C" long WINAPI ems_plugin_version(
short FAR* api_version      /* Out: Place to return api version */
);
```

Eudora calls this function once when it is loading the plug-in to determine what version of the API it implements. The API version that should be returned is defined in the API include files as `EMS_VERSION`.

| Macintosh Eudora | EMS-API Version |
|---|---|
| v3.0 | v1 |
| v3.1-> v3.x | v1, v3 |
| v4.x | v1, v3, v4 |

| Windows Eudora | EMS-API Version |
|---|---|
| v3.0 | v2 |
| v3.1-> v3.x | v2, v3 |
| v4.x | v2, v3, v4 |

On the Macintosh, Eudora checks the `version` string in the `thng` resource as it is loading the plug-in.

#### Parameters

← *apiVersion*

Put the version of the Plug-in's API.

#### Return Value

`EMSR_OK`: All is OK, Eudora will continue loading plug-in.

Anything else: Eudora will unload the plug-in and not call any of its functions.

### 8.8. Initialize plug-in and get its basic info

Macintosh:

```
pascal long ems_plugin_init(
    Handle globals,             /* Out: Return for allocated instance structure */
    short eudoraAPIVersion,     /* In: the Version of the API Eudora is using */
    emsMailConfigP mailConfig,  /* In: Eudora mail configuration */
    emsPluginInfoP pluginInfo   /* Out: Return Plugin Information */
);
```

Windows:

```
extern "C" long WINAPI ems_plugin_init(
    void FAR * globals,         /* Out: Return for allocated instance structure */
    short eudoraAPIVersion,     /* In: the Version of the API Eudora is using */
    emsMailConfigP mailConfig,  /* In: Eudoras mail configuration */
    emsPluginInfoP pluginInfo   /* Out: Return Plugin Information */
);
```

This function is called once by Eudora as the plug-in is loaded. It is a good place to do plug-in specific initializations.

#### Parameters

*globals*

Return here the pointer to globals that will be passed back in the rest of the functions. This should be used for global data in the plug-in scope.

For the Macintosh, the globals argument is a handle to a data structure holding the plug-in's global state. It is passed to all functions. The Component Manager takes care of carrying this between calls. If the plug-in is authored using SDK component main, `ems-component.c`, then this structure should be defined in `usertrans.h`.

For Windows, the `ems_plugin_init` function must allocate this storage and return a pointer to it in the location pointed to by the globals parameter. Eudora will then pass this pointer into all other translation API calls for that plug-in-in. It should be de-allocated in the `ems_plug_in_finish` function.

*eudoraAPIVersion*

The version of the API Eudora is using.

*mailConfig*

*size* *sizeof(emsMailConfig)*

*eudoraWnd* **[Windows only]**

A pointer to Eudora's main application window.

*configDir*

The path of a folder in which is the suggested location for a plug-ins own configuration data. This will be the users mail directory + the plug-ins directory. This path varies as the Eudora folder and setting path varies, thus a plug-in's settings will vary with the Eudora settings if the user has multiple Eudora set ups on the system.

➔ *userAddr*

The *userAddr -> realname* is the user's human name as entered in the "Real Name" setting of the dominant personality. The *userAddr -> address* is the rfc-822 address the user has configured as their return address, or if no return address has been configured, it is the POP account of the dominant personality.

*pluginInfo*

➔ *size* `sizeof(emsPluginInfo)`

➔ *numTrans*

The total number of translators in this plug-in. Translator IDs range from 1 to *numTrans*.

➔ *numAttachers*

The total number of special menu items in this plug-in. IDs range from 1 to *numAttachers*.

➔ *numSpecials*

The total number of special menu items in this plug-in. IDs range from 1 to *numSpecials*.

➔ *desc*

A short string suitable for a splash or about screen and should include the plug-in version number. As with all strings returned to Eudora, on the Mac it must be allocated with `NewHandle()` and on Windows with the plug-ins internal memory allocator.

➔ *id*

Each plug-in must have a unique ID number and return it in the `plugin_id` parameter. These are available from an email auto-responder by sending a message to <emsapi-ids@qualcomm.com>. See section 2.3 for more details on the auto-responder.

➔ *icon*

The icon is shown in the plug-ins about box. On the Macintosh it should be an icon suite allocated with `NewHandle()`. For Windows it should be a 32x32 `HICON` allocated with the plug-ins own allocator function.

➔ *mem_rqmnt*

The memory footprint required to run this plugin (Mac).

### Return Value

`EMSR_OK`: All is OK, Eudora will continue loading plug-in.

Anything else: Eudora will unload the plug-in and not call anymore of its functions.

### 8.9. Get basic translator info

Macintosh:

```
pascal long ems_translator_info(
    Handle globals,              /* In: Pointer to plugin instance structure */
    emsTranslatorP transInfo     /* In/Out: Return Translator Information */
);
```

Windows:

```
extern "C" long WINAPI ems_translator_info(
    void FAR * globals,          /* In: Pointer to plugin instance structure */
    emsTranslatorP transInfo     /* In/Out: Return Translator Information */
);
```

This function is called for each translator ID by Eudora as it builds its internal lists of translators while it starts up. Note that any of the pointers to places to return data may be NULL so Eudora does not have to request all the details at once. Some items like the flags and types will be loaded once initially, while others such as the icon may be retrieved each time it is needed.

#### Parameters

*globals*

The pointer to the globals is passed back for the translator to use.

*transInfo*

*size* `sizeof(emsTranslator)`

*id*

The `id` selects the particular translator in the plug-in for which the data is to be returned.

*type*

This describes what type of translator this is (e.g., EMST_LANGUAGE), it must be one of the types that start as `EMST_` .

*flags*

The contexts in which a translator can be called. Multiple flags are bitwise or-ed together.

If `EMSF_Q4_COMPLETION` is set, `EMSF_DEFAULT_Q_ON` will default the translator to on. Set `EMSF_TOOLBAR_PRESENCE` to have this on the main toolbar by default. Set `EMSF_ALL_TEXT` in conjunction with `EMSF_ON_REQUEST` so you'll get the whole message instead of just the selection.

*desc*

The description is a short string that is used for pull-down menu items. It is the only thing that identifies a translator on the menu so it should include something that indicates which plug-in it belongs to. An example might be "AcmeTrans Spanish-English."

*icon*

The icon is used for presentation to the user in several places. On the Macintosh an icon suite should be returned and should be allocated using `NewHandle()`. For Windows, the icon should be a 16x16 `HICON` allocated with the plug-in's memory allocator.

### Return Value

EMSR_OK: All is OK, Eudora will continue load up the translator.

Anything else:  Error will be logged.

### 8.10. Check to see whether a translation can be performed

Macintosh:

```
pascal long ems_can_translate_file(
    Handle globals,              /* In: Pointer to plugin instance structure */
    emsTranslatorP trans,        /* In: Translator Info */
    emsDataFileP inTransData,    /* In: What to translate */
    emsResultStatusP transStatus /* Out: Translations Status information */
);
```

Windows:

```
extern "C" long WINAPI ems_can_translate(
    void FAR * globals,          /* In: Pointer to plugin instance structure */
    emsTranslatorP trans,        /* In: Translator Info */
    emsDataFileP inTransData,    /* In: What to translate */
    emsResultStatusP transStatus /* Out: Translations Status information */
);
```

This function checks to see whether a data item can be translated. It is called by Eudora before every translation is attempted and in some cases to determine whether a translation can be performed in a later context on some data. The `trans->id` specifies which translator from the plug-in is being called. The `inTransData->context` parameter is a `long` with only one bit set to indicate the context (e.g., : `EMSF_ON_ARRIVAL,` or `EMSF_Q4_TRANSMISSION`). The MIME type of the input data is always provided in the `inTransData` parameter.

#### Parameters

*globals*

The pointer to the globals is passed back for the translator to use.

*trans*

- *size* `sizeof(emsTranslator)`
- *id*

  The `id` selects the particular translator in the plug-in for which the data is to be returned.

- *properties*

  Only used when in the `EMSF_Q4_TRANSMISSION` context. `ems_queued_properties` can set this.

*inTransData*

- *size* `sizeof(emsTranslator)`
- *context*

  This is a long with only one bit set that represents the current context (e.g.,: `EMSF_ON_ARRIVAL`, or `EMSF_Q4_TRANSMISSION`)

- *info*

  The MIME type of the input data. This is what should be checked to see if the translator wants to translate this message.

*header*

➔ *size* `sizeof(emsHeaderData)`

➔ *to*

➔ *from*

➔ *subject*

➔ *cc*

➔ *bcc*

These fields will be populated when `EMSF_BASIC_HEADERS` is set for the translator. They are read only.

➔*rawHeaders*

This field will be populated with the message headers when `EMSF_ALL_HEADERS` is set for the translator. The header is in canonical MIME format, so each line is delimited by a carriage return-linefeed pair. This is read-only information.

*transStatus*

➔ *size* `sizeof(emsResultStatus)`

➔ *error*

If error is returned, Eudora will display this in a error dialog. If there was no error, set to NULL.

➔ *code*

Return for translator-specific result code

### Return Value

`EMSR_NOW`: The translator will translate this message. `ems_translate_file` will be called next.

`EMSR_NOT_NOW`: The translator will translate this message, but not now. When writing an ON_DISPLAY translator, when receiving the message ON_ARRIVAL, check to see if this is a message that this plug-in can translate later, then return EMSR_NOT_NOW so it will be called in the ON_DISPLAY context.

`EMSR_CANT_TRANS:` This is not a message that this translator can translate.

Anything else: Failure. This will cause Eudora to put up an error message associated with the return. Fill in `transStatus->error` if you want Eudora to display an error. `EMSR_OK` is considered a failure return.

### 8.11. Performing translations

Macintosh:

```
pascal long ems_translate_file(
    Handle globals,              /* In: Pointer to plugin instance structure */
    emsTranslatorP trans,        /* In: Translator Info */
    emsDataFileP inFile,         /* In: What to translate */
    emsProgress progress,        /* Func to report progress/check for abort */
    emsDataFileP outFile,        /* Out: Result of the translation */
    emsResultStatusP transStatus /* Out: Translations Status information */
);
```

Windows:

```
extern "C" long WINAPI ems_translate_file(
    void FAR * globals,          /* In: Pointer to plugin instance structure */
    emsTranslatorP trans,        /* In: Translator Info */
    emsDataFileP inFile,         /* In: What to translate */
    emsProgress progress,        /* Func to report progress/check for abort */
    emsDataFileP outFile,        /* Out: Result of the translation */
    emsResultStatusP transStatus /* Out: Translations Status information */
);
```

This function performs the actual translation. Note that `ems_can_translate` is always called by Eudora before this function is called so the translator author need not make the same checks here. This function will only be called if `ems_can_translate` returns `EMSR_NOW`.

The translator may behave different ways in different contexts. For example when verifying a signature in the automatic on-display context, it may choose to fail if the certificate necessary to verify is unavailable, but in the on-request context it may prompt the user to locate the certificate.

For translations on message text, the temporary files are deleted immediately after the translation is complete. Attachments, however are not deleted until the user removes them. *This will change when Eudora switches to using MIME storage internally.*

#### Parameters

*globals*

The pointer to the globals is passed back for the translator to use.

*trans*

- *size* `sizeof(emsTranslator)`
- *id*

  The `id` selects the particular translator in the plug-in for which the data is to be returned.

- *properties*

  Only used when in the `EMSF_Q4_TRANSMISSION` and `EMSF_Q4_COMPLETION` context. `ems_queued_properties` can set this.

*inFile*

- *size* `sizeof(emsTranslator)`
- *context*

  This is a long with only one bit set that represents the current context (e.g.,: `EMSF_ON_ARRIVAL`, or `EMSF_Q4_TRANSMISSION`)

➔ *info*

The MIME type of the input data. This is what should be checked to see if the translator wants to translate this message.

*header*

➔ *size* `sizeof(emsHeaderData)`

➔ *to*

➔ *from*

➔ *subject*

➔ *cc*

➔ *bcc*

These fields will be populated when `EMSF_BASIC_HEADERS` is set for the translator.

➔ *rawHeaders*

This field will be populated with the message headers when `EMSF_ALL_HEADERS` is set for the translator.

➔ *fileName*

The file to be translated. If `EMSF_REQUIRES_MIME` is set `transInfo->flag` `ems_translator_info` is called, all the headers will be supplied in the file. If this is the top most part, all the top most headers will be there, if this is a part, only the part's headers will be there.

*progress*

The translator should call the function periodically with an argument between 0 (just begun) and 100 (complete) to indicate its progress. The translator should check the return value from the function. If the value is 1 it should abort the translation, and if 0 it should continue. A translator may display its own progress status and not make use of the one which Eudora supplies. It should still call the progress function periodically with an argument of -1 to check for an abort. If the call to the progress function returns 1 indicating abort at any time, the translation must be aborted. In other words, the abort indication must never be ignored.

*outFile*

➔ *size* `sizeof(emsTranslator)`

➔ *info*

The translator must always return the correct MIME type of the translation output in this parameter even if the translator generates MIME. Thus, if the translator is unwrapping a MIME object it must parse the `Content-Type:` header and return its value in `out_mime`. This also implies that translators that generate MIME will return the resulting output MIME type in two places, in the actual data and in the `out_mime` parameter.

Except for translations in the on-request context, the input and output MIME types must be different in order to avoid an infinite translation loop. This can be done by adding a MIME parameter to the MIME type to indicate a translation has been performed. A good parameter name is `x-eudora-translated`, and a good value is the name of the translator and the context (e.g., `spanish-english-on-arrival`). Such a parameter will be ignored by all other MIME parsing. The translator should check for this parameter in its `ems_can_translate` function.

➔ *fileName*

An empty output file is created by Eudora, and the name of this file is passed into the translator. The translator should write its output data into the file. If the translation is aborted Eudora will clean up and remove this file.

*transStatus*

➔ *size* `sizeof(emsResultStatus)`

➔ *desc*

If `desc` is returned it will be displayed in the message window adjacent to the entity just translated along with some visual indication that it is tied to the entity.

➔ *error*

If error is returned, Eudora will display this in a error dialog. If there was no error, set to NULL.

➔ *code*

For most translations the `out_code` is ignored, but for translations of type `EMST_SIGNATURE` it should be one of the constants `EMSC_SIGOK`, `EMSC_SIGBAD`, or `EMSC_SIGUNKNOWN` to indicate the status of the signature. Eudora displays the bar that ties the icon and status message to the translated text differently, depending on the result of the signature verification.

### Return Value

`EMSR_OK`: The translator will translate this message. `ems_translate_file` will be called next.

`EMSR_DATA_UNCHANGED:` Eudora will leave the original text in the message and ignore the returned outFile data. Only applicable in the on-request state. In other states, this will be treated as an error.

Anything else: Failure. This will cause Eudora to put up an error message associated with the return. Fill in `transStatus->error` if you want Eudora to display an error.

## 8.12. Finish use of a plug-in

Macintosh:

```
pascal long ems_plugin_finish(
     Handle globals              /* In: Pointer to plugin instance structure */
);
```

Windows:

```
extern "C" long WINAPI ems_plugin_finish(
     void FAR* globals           /* In: Pointer to plugin instance structure */
);
```

This gives the plug-in a chance to free allocated memory, save state information, etc. Windows translators should de-allocate the globals memory, but Macintosh translators should not.

### Parameters

➔ *globals*

The pointer to the globals is passed for clean up.

### Return Value

`EMSR_OK`: All is OK.

Anything Else: Eudora will log an error.

### *8.13. Free API data structures (Windows only)*

```
extern "C" long WINAPI ems_free(
    void FAR* mem               /* Memory to free */
);
```

This is called by Eudora to free data structures passed from a plug-in to Eudora. This data includes strings, addresses, and the MIME type data structure. This is not used on the Macintosh since all data on it are Handles allocated with standard functions.

#### Parameters

➔ *mem*

The pointer to the memory is passed for clean up.

#### Return Value

EMSR_OK: All is OK.

Anything Else:  Eudora will log.

### 8.14. Plug-in Settings Dialog

```
Macintosh:
pascal long ems_plugin_config(
    Handle globals,             /* In: Pointer to plugin instance structure */
    emsMailConfigP mailConfig   /* In: Eudora mail info */
);

Windows:
extern "C" long WINAPI ems_plugin_config(
    void FAR  globals,          /* In: Pointer to plugin instance structure */
    emsMailConfigP mailConfig   /* In: Eudora mail info */
);
```

The icon and name of the plug-in will appear in a plug-ins "Installed Message Plug-ins" dialog selected from the "Message Plug-ins Settings" item under the "Special" menu. When the user selects a plug-in and clicks the "Settings…" button, this function will be called. The plug-in should put up its settings panel, interact with the user and store the result.

After this function is called, Eudora will call `ems_trans_info` for each translator to see if flags have changed.

#### Parameters

➔ *globals*

The pointer to the globals is passed back for the translator to use.

*mailConfig*

➔ *configDir*

The path of a folder in which is the suggested location for a plug-ins own configuration data. This will be the users mail directory + the plug-ins directory. This path varies as the Eudora folder and setting path varies, thus a plug-in's settings will vary with the Eudora settings if the user has multiple Eudora set ups on the system.

➔ *userAddr*

The *userAddr -> realname* is the user's human name as entered in the "Real Name" setting of the dominant personality. The *userAddr -> address* is the rfc-822 address the user has configured as their return address, or if no return address has been configured, it is the POP account of the dominant personality.

#### Return Value

`EMSR_OK`: All is OK.

Anything Else: Eudora will log.

### 8.15. Queued translation properties

```
Macintosh:
pascal long ems_queued_properties(
    Handle globals,             /* In: Pointer to plugin instance structure */
    emsTranslator trans         /* In/Out: The translator */
    long *selected              /* In/Out: state of this translator */
);

Windows:
extern "C" long WINAPI ems_queued_properties(
    void FAR * globals,         /* In: Pointer to plugin instance structure */
    emsTranslator trans         /* In/Out: The translator */
    long *selected              /* In/Out: state of this translator */
);
```

For queued translations the user selects the translation possibly including some parameters about it, at a different time than the translation is performed. This function allows the parameters to be stored with the message while it is in the queue.

This function is optional. If it is not supplied, queued translations will be toggled on and off automatically by Eudora. If this is function is present it will be called when the user clicks the icon in the composition bar. The function is passed the usual parameters to identify the translator and context. When called, this function may put up a dialogue and interact with the user.

If the user has selected `EMST_PREPROCESS` and `EMST_SIGNATURE` translations, and an `EMST_COALESCED` translation is available, it will be called instead as described previously. The properties of the two translators will be passed to the `EMST_COALESCED` translator concatenated and separated by a comma. The `EMST_SIGNATURE` translator's parameters will be first. This way nothing special need be done by the translators a queue time. They each set their parameters as they wish.

#### Parameters

➔ *globals*

The pointer to the globals is passed back for the translator to use.

*trans*

➔ *size* `sizeof(emsTranslator)`

➔ *id*

The id selects the particular translator in the plug-in for which the data is to be returned.

➔ *properties*

These properties will get stored with the message only if `selected` is set. It will be passed back the actual translation is performed in the `ems_translate_file` function. The string must be printable ASCII characters from "!" (0x21) to "~" (0x7e) and must not contain any commas (0x2c). The string must also be less than 100 bytes.

➔➔ *selected*

Eudora will pass the current selection state. Return whether is should be selected or not.

#### Return Value

`EMSR_OK`: All is OK

Anything Else: Eudora will log.

### 8.16. Attachment Menu Items

```
Macintosh:
pascal long ems_attacher_info(
    Handle globals,              /* In: Pointer to plugin instance structure */
    emsMenuP attachMenu          /* Out: The menu */
);

Windows:
extern "C" long WINAPI ems_attacher_info(
    void FAR * globals,          /* In: Pointer to plugin instance structure */
    emsMenuP attachMenu          /* Out: The menu */
);
```

Eudora will place these menus in the Message ➔ Attach sub-menu. When a user selects an attachment plug-in, the `ems_attacher_hook` function will be called.

#### Parameters

➔ *globals*

The pointer to the globals is passed back for the translator to use.

*attachMenu*

➔ *size* `sizeof(emsMenu)`

➔ *id*

ID of translator to get information for.

➔ *desc*

The text that will go in the Message->Attachment-> sub-menu.

➔ *icon*

A 16x16 icon that will show up in the menu and in the custom toolbar selection. NULL will display a default icon

➔ *flags*

Set `EMSF_TOOLBAR_PRESENCE` so this will show up on the main toolbar on startup.

#### Return Value

`EMSR_OK`: All is OK.

Anything Else: Eudora will not load up the item.

### 8.17. Attachment Menu Hook

```
Macintosh:
pascal long ems_attacher_hook(
    Handle globals,              /* In: Pointer to plugin instance structure */
    emsMenuP attachMenu,         /* In: The menu */
    FSSpec *attachDir,           /* In: Location to put attachments */
    long *numAttach,             /* Out: Number of files attached */
    emsDataFileH *attachFiles    /* Out: Name of files written */
);

Windows:
extern "C" long WINAPI ems_attacher_hook(
    void FAR * globals,         /* In: Pointer to plugin instance structure */
    emsMenuP attachMenu,        /* In: The menu */
    LPSTR attachDir,            /* In: Location to put attachments */
    long * numAttach,           /* Out: Number of files attached */
    emsDataFileP ** attachFiles /* Out: Name of files written */
);
```

When a user selects an attacher, the `ems_attacher_hook` function will be called. The plug-in can create a user interface to select or create a file. The path to this file should be returned.

#### Parameters

➔ *globals*

The pointer to the globals is passed back for the translator to use.

*attachMenu*

➔ *size* `sizeof(emsMenu)`

➔ *id*

ID of translator to get information for.

➔ *desc*

The text that will go in the Message->Attachment-> sub-menu.

➔ *attachDir*

The suggested directory to put the attached file. If the file is put into this directory, Eudora will manage when the file is deleted.

➔ *numAttach*

The number of files that will be attached.

*AttachFiles (this is an array of Attached files, so 'n' files can be attached)*

➔ *size* `sizeof(emsTranslator)`

➔ *fileName*

The file to be attached

#### Return Value

`EMSR_OK`: All is OK. `AttachFile` must contain a path to a file as well.

Anything Else:  Eudora will log an error.

### 8.18. Special Menu Items

```
Macintosh:
pascal long ems_special_info(
    Handle globals,              /* In: Pointer to plugin instance structure */
    emsMenuP specialMenu         /* Out: The menu */
);

Windows:
extern "C" long WINAPI ems_special_info(
    void FAR * globals,         /* In: Pointer to plugin instance structure */
    emsMenuP specialMenu         /* Out: The menu */
);
```

Eudora will place these menus in the Tools (Windows) or Special (Macintosh) menu. When a user selects an attacher menu item, the `ems_special_hook` function will be called.

#### Parameters

➔ *globals*

The pointer to the globals is passed back for the translator to use.

*attachMenu*

➔ *size* `sizeof(emsMenu)`

➔ *id*

ID of translator to get information for.

➔ *desc*

The text that will go in the menu item.

➔ *icon*

A 16x16 icon that will show up in the menu and in the custom toolbar selection. NULL will display a default icon.

➔ *flags*

Set `EMSF_TOOLBAR_PRESENCE` so this will show up on the main toolbar on startup.

#### Return Value

`EMSR_OK`: All is OK.

Anything Else: Eudora will not load up the item.

### 8.19. Special Menu Hook

```
Macintosh:
pascal long ems_special_hook(
    Handle globals,             /* In: Pointer to plugin instance structure */
    emsMenuP specialMenu        /* In: The menu */
);

Windows:
extern "C" long WINAPI ems_special_hook(
    void FAR * globals,         /* In: Pointer to plugin instance structure */
    emsMenuP specialMenu        /* In: The menu */
);
```

This will be called the special menu item is selected by the user.

#### Parameters

➔ *globals*

The pointer to the globals is passed back for the translator to use.

*attachMenu*

➔ *size* *sizeof(emsMenu)*

➔ *id*

ID of translator to get information for.

#### Return Value

`EMSR_OK`: All is OK.

Anything Else: Eudora will log an error.

# 9. Changes in latest API descriptions

### *December 1997*

- Updated to V4
- `EMSF_Q4_COMPLETION` supported for outgoing messages
- Translator Icons will be displayed for Attachers, Special Tools, struct emsMenu now includes icon and flags fields.
- `EMSF_TOOLBAR_PRESENCE` will default translator icons on main toolbar
- Non-Mime messages will be sent to translators as text/plain for the `ON_ARRIVAL` context.
- `ON_REQUEST` translators will now get the option of handling text/html or text/plain `EMSF_ALL_TEXT` will give all the text to `ON_REQUEST` translators

### *August 1997*

- Finalized v3
- Complete format overhaul to elliminate unwanted fonts and styles
- Removed discussion of future features which do not apply to v4
- Revised figure 1
- Appendix B - MIME Type Mappings added
- Section 6: Attacher Plug-ins revised

### *December 1996*

- Updated to V3
- Parameter Blocks passed into functions instead of parameter lists
- ems_attacher_info, ems_attacher_hook
- ems_special_info, ems_special_hook
- removed translator subtype
- access to all headers
- EMSR_UNCHANGED allows for translators that don't change data
- access to content-disposition

### *August 20, 1996*

- ***Incremented API version number to 2***
- Implemented the settings dialogue
- Implemented queued_properties
- Added properties parameter to `ems_can_translate`(), `ems_translate_file`() and `ems_translate_buf`()
- Added user name, address and configuration folder to `ems_plugin_init`() call
- Changed name of `ems_can_translate_file`() to `ems_can_translate`() and removed a couple of parameters.

### *July 19, 1996*

- Clarified features in version 1 vs. future versions
- Completed name change from tlapi to ems api
- Added description of ID allocating auto responder
- Major clarifications to use of MIME format and type
- Added about box to list loaded plug-ins
- Clarifications on the translation process
- More consistent terminology and notation
- Specifies Windows icon format

- Specifies Windows plug-in search directories
- Abort return code added, plug-ins required to abort when told to do so
- Moved MIME background to an appendix
- Dropped the buffer version of ems_can_translate

### *May 22, 1996*

- Removed `DOES_MIME_LEAVES` since it was unused and meaningless
- Progress function now works.
- Described some future additions
- on-request translators now checks MIME types
- More documentation clarifications and rewording (MIME-related stuff)
- Described planned implementation of buffer-based translation
- Significant support for Windows added (but Windows SDK isn't available yet)
- Windows allocator function added

### *April 1996*

- Switch to separate Macintosh and Windows API definitions
- Removed OP code and lookup function
- Added calling interface details for Mac and Windows
- Added export warning for translation authors
- Page numbering and minor wording changes
- Major clarifications
- Added module_version function
- Removed de-allocator and version arguments from `module_init`
- Added module icon argument to `module_init`

# 10. References


[Component] *Inside Macintosh: More Macintosh Toolbox*. Addison Wesley 1993.

[DLL] *Windows SDK that describes DLL's*

[Crocker] CROCKER, D. *Standard for the format of ARPA Internet Text Messages*. Internet Engineering Task Force, RFC 822. 1982.

[DISP] DORNER, S. AND TROOST, R. *Commuinicating Presentation Information in Internet Messages: The Content-Disposition Header*. Internet Engineering Task Force, RFC 1806.

[MIME] BORENSTEIN, N., FREED, N., KLENSIN, J., MOORE, K., AND POSTEL, J. *MIME: Multipart Internet Mail Extensions*. Internet Engineering Task Force, RFC 2045, 2046, 2047, 2048

[FREED] FREED, NED, ET AL. *Security Multipart for MIME: multipart/signed and multipart/encrypted.* Internet Engineering Task Force, RFC 1847. 1995

[Lang] ALVESTRAND, HARALD. *Tags for Identifcation of Langages.* Internet Engineering Task Force, RFC 1766. 1995

[Enriched] RESNICK, PETE AND WALKER, AMANDA. *RFC-1896, The text/enriched content type.* Internet Engineering Task Force, RFC 1896. 1996

# Appendix A - A brief introduction to MIME

MIME (Multipart Internet Mail Extensions) [MIME] is the Internet standard for describing objects in Internet e-mail. It is also used in other applications on the Internet such as the World-Wide Web. The MIME standard has three main functions. It provides type tagging information for e-mail messages and their parts. It provides a format for representing object types and message structure, and it provides transfer encoding for safely passing 8bit text and binary data through 7bit text-only data paths.

```
Content-Type: MULTIPART/MIXED; BOUNDARY="-559023410-851401618-831602781=:25682"

---559023410-851401618-831602781=:25682
Content-Type: TEXT/PLAIN; charset=US-ASCII

This is a little text part of the message

---559023410-851401618-831602781=:25682
Content-Type: IMAGE/GIF; charset=US-ASCII
Content-Transfer-Encoding: BASE64
Content-Disposition: attachment; name="apipict.gif"

IC8qID09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09PT09
eHRlbmRlZCBNZXNzYWdlIFNlcnZpY2VzIEFQSSBTREsgMS4wYjIgKE1heSB4
eCAxOTk2KQ0KICAgIFRoaXMgU0RLIHN1cHBvcnRzIEFQSSB2ZXJzaW9uIDEN
CiAgICBDb3B5cmlnaHQgMTk5NSwgMTk5NiBRVUFMQ09NTSBJbmMuDQogICAg

---559023410-851401618-831602781=:25682--
```

A small example of a MIME e-mail message is shown above. It is a two-part message with the first part being some text and the second part being an attachment. The second part is a GIF image with base64 encoding so the binary GIF image can be passed through 7-bit channels.

Each MIME type has a top level type, a sub-type and optional parameters. The top-level content types are relatively fixed and currently number seven: text, application, multipart, message, image, audio and video. The multipart type is of particular importance because it is a container for any number of MIME objects, thus MIME allows nested structuring of message objects. There are many sub-types for each top-level type. New sub-types can be registered as long as there is a document giving a basic description of them. The actual type information is usually expressed as ASCII text in the form `type/sub-type`. The type may also include parameters which allow specification of further details about the types. The set of parameters is completely dependent on the sub-type, though some are common to more than one sub-type. Two common parameters are character-set and language.

In addition to defining a typing scheme, MIME very precisely specifies data formats for representing the type data and for creating a data object that combines the actual content data and the type information.

Because MIME objects are commonly transferred via Internet e-mail, often a 7bit text-only path, the MIME standard also includes an encoding scheme for expressing arbitrary data as 7bit text with limited line lengths. This is known as “content transfer encoding”.

Because MIME is used to pass objects over the network between unlike computing platforms (e.g., Macintosh and Windows), it defines a notion of a canonical format for data objects. This is a format for a data object of a specific type that is either common to the platforms it is used on, or is defined to be the interchange format for the object across platforms. The most important canonical format is for text objects because the end-of-line delimiter for text files varies between major computing platforms. Canonical text in MIME messages has lines separated by the CRLF (carriage return and line feed) pair and does not include CR or LF except at the end of a line.

A canonical on-the-wire formatted MIME entity is an octet stream (which may be in the process of being transmitted, in a file on disk, or in a memory buffer) representing message objects in their canonical format tagged with MIME types.

It possible to define proprietary MIME types for specific translator applications. It is also possible to go through the standards process to define new MIME types to be used widely on the Internet. The types enable translators to easily and efficiently recognize data on which they wish to operate.

The reader is referred to the MIME standards documents [MIME] for further details.

# Appendix B - MIME type mappings

When a file is attached either manually or via an EMS API attacher plug-in, Eudora tries to find the correct MIME type/subtype for the attachment. If the MIME type/subtype cannot be found, the default "application/octet-stream" is used.

Each platform has its own method for determining the MIME type of a file.

***Windows***

Under Windows, the MIME type of the file is determined by the file extension. There is a section of the EUDORA.INI file which maps many common file extensions to their MIME type/subtype. Within the EUDORA.INI file, the "MAPPINGS" section contains all the extension to MIME mappings.

Each line in the MAPPINGS section has the following format:

```
<direction>=<extension>,<Mac creator>,<Mac type>,<MIME type>,<MIME subtype>
```

An example section:

```
[Mappings]
out=txt,ttxt,TEXT,text,plain
both=doc,MSWD,,application,msword
in=xls,XCEL,,,
```

The direction specifies when the given mapping should be applied. This field can be either "in", "out", or "both." Messages received by Eudora are processed by the "in" and "both" mappings. Composed messages being sent are processed by the "out" and "both" mappings.

Note that every mapping line has exactly four commas, regardless of how much information is provided.

***Macintosh***

On the Macintosh, MIME mappings for attachments are controlled by resources inside the Eudora application of type 'EuIM' and 'EuOM'. These resources relate the MIME format's content type and sub-type with file extensions and Macintosh type and creator codes. These resources also contain flags which specify whether the attachment is text, a basic type, and other properties.