This chapter describes how you can use the Translation Manager to direct the translation of documents from one format to another. This chapter also gives an overview of Macintosh Easy Open. Macintosh Easy Open uses the Translation Manager to provide extensive data translation services for Macintosh computers. Macintosh Easy Open uses the Translation Manager to provide

- automatic translation of a document from one format to some other format if the application that created it is not available when the user attempts to open the document
- automatic translation of documents drop-launched onto an application
- enhanced Standard File Package file-opening dialog boxes and (when necessary) automatic translation of documents the user selects in those dialog boxes
- batch desktop translation of documents
- automatic translation of data pasted from the Clipboard
- automatic translation of data in editions

Most applications take advantage of these services automatically if they use the system software to open documents. You can, however, enhance your application's interaction with Macintosh Easy Open by adding several resources to your application's resource file. For example, the Finder and Macintosh Easy Open work with the Standard File Package to list in the file opening dialog box all files that your application can open, including those that it can open only after they have been translated from their current format to another format. See "Declaring the File Types Your Application Can Open" on page 7-13 for instructions on adding the required resources to your application. If, however, your application doesn't use the Standard File Package when opening files, you might also need to use the Translation Manager to direct the translation. See "Translating Files Explicitly" on page 7-17 for more information.

This chapter also describes how to write a translation extension. Macintosh Easy Open doesn't do any translating itself; instead, it uses translation extensions to translate documents (data in files) and scraps (data in memory) in the situations listed above. Translation extensions also need to be able to report the kinds of files or scraps they can handle and to identify specific files that need to be translated. You're likely to need to write a translation extension only if you are developing file or scrap translators (also known as convertors or filters).

Macintosh Easy Open and the Translation Manager are not available in all system software versions. You should use the `Gestalt` function to ensure that the services you need are available before calling them. See "Checking for the Translation Manager" on page 7-12 for details.

To use this chapter, you should already be familiar with the Standard File Package, the Scrap Manager, the Edition Manager, Finder-related resources, and the Component Manager. For information on the Standard File Package, see *Inside Macintosh: Files*. For information on the Scrap Manager and the Component Manager, see their corresponding chapters in this book. For information on the Edition Manager, see *Inside Macintosh: Interapplication Communication*. For information on Finder-related resources, see *Inside Macintosh: Macintosh Toolbox Essentials*.

About the Translation Manager

The Translation Manager provides extensive data translation services for Macintosh computers. **Macintosh Easy Open** uses the Translation Manager to provide four basic services:

- translation of documents opened from the Finder
- automatic translation of documents opened by applications that use the Standard File Package
- batch translation of documents at the desktop level
- automatic translation of data in editions or pasted from the Clipboard

These services allow your application to open documents created by other applications (possibly running on other operating systems) and to import data from other applications with better fidelity than previously possible.

Macintosh Easy Open provides the services that the Finder and the Standard File Package use to implement **implicit translation** (the conversion of a file or scrap without direct intervention from the application). The Finder needs to know which applications are capable of opening a document, either directly or after the document has been translated to another file format. The Standard File Package needs to know which other file types can be translated to some file type that the application can read. Both the Finder and the Standard File Package then call Macintosh Easy Open to translate a file to another format.

Macintosh Easy Open does not do any translating itself, and it does not have any knowledge of translation data models. Instead, it delegates these functions to translation extensions or to applications with built-in translation capability. Translation extensions and application translation capabilities operate as “black boxes” to Macintosh Easy Open.

A **translation extension** is responsible for many things, including recognizing and translating files or scraps. A translation extension might be a complete entity, able to recognize and translate all by itself. Other translation extensions might require external files, usually called translators or filters, to perform their work. In either case, the whole is called a **translation system.**

At system startup (or whenever new translation extensions become available), Macintosh Easy Open catalogs the translation capability of each translation extension and each application, and then invokes each as needed. Macintosh Easy Open can support multiple translation systems.

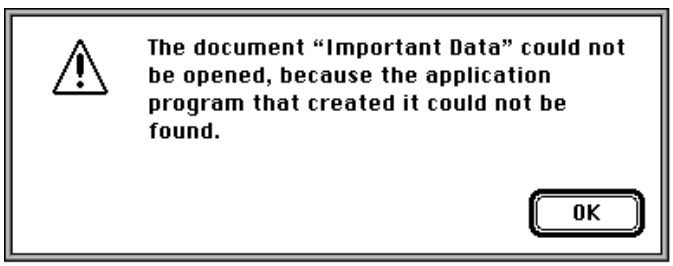
There are two types of translation systems: file translation systems and scrap translation systems. A **file translation system** can translate from one file format to another. A **scrap translation system** can translate buffers in memory. Macintosh Easy Open distinguishes between the two because a file format in memory might differ from the same file format on disk. A single translation system, however, might contain both kinds of translators.

The following four sections describe in greater detail the capabilities of Macintosh Easy Open and its interactions with other pieces of the Macintosh system software.

Opening Documents From the Finder

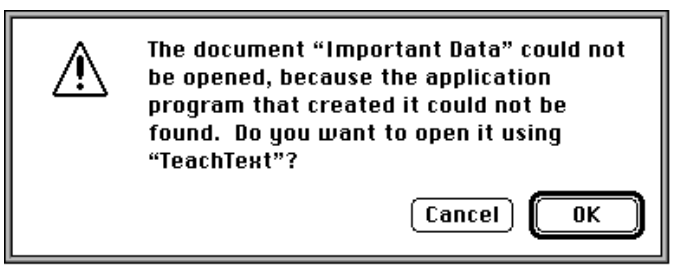
A user can ask the Finder to open a document in several ways, for example, by selecting the document's icon and choosing the Open command in the Finder's File menu or (more typically) by double-clicking the document's icon. If Macintosh Easy Open is not present in the operating environment and the user attempts to open a document created by an application that isn't available, the Finder displays the alert box shown in Figure 7-1.

Figure 7-1 The Finder's application-unavailable alert box



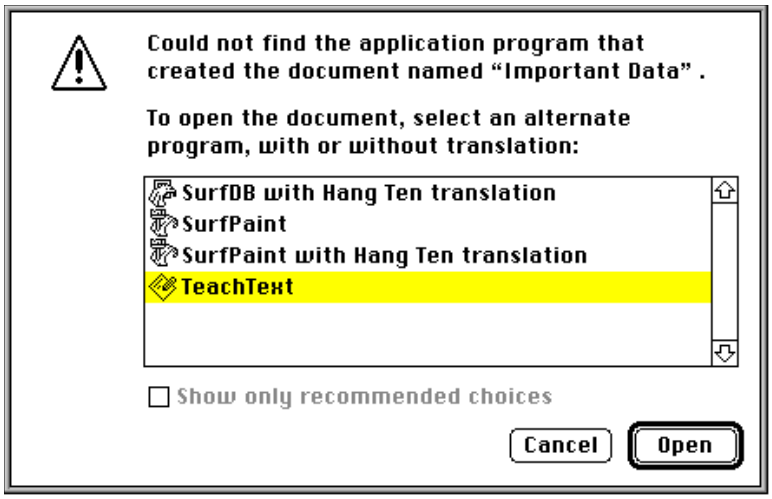
If the document the user wants to open is of type `'TEXT'` or `'PICT'` and the creator application cannot be found, the Finder instead displays the alert box shown in Figure 7-2, which allows the user to try to open the document using the TeachText application.

Figure 7-2 The application-unavailable alert box for `'TEXT'` and `'PICT'` documents



When Macintosh Easy Open is available, it intercedes in the Finder's document-opening process. For example, if the user attempts to open the document "Important Data" (of type `'SURF'`) created by the SurfWriter application and that application isn't available on the user's system, the Finder displays a dialog box like the one shown in Figure 7-3. This dialog box contains a list of all applications that can open a document of that type.

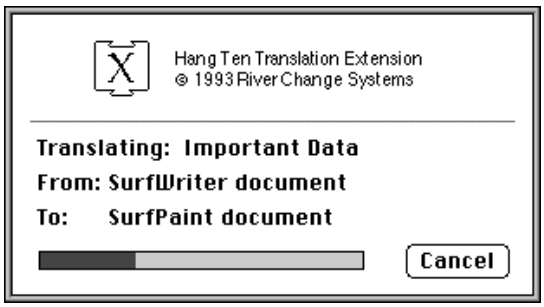
Figure 7-3 The translation choices dialog box



In this dialog box, the user can select a translation path from the document's current format to one that can be opened by some application that is available. In this way, the user can open documents created by missing or unavailable applications.

Macintosh Easy Open lists two kinds of applications in the dialog box shown in Figure 7-3, applications that can open the file natively (that is, in its current format) and those that can open the document only after the document has been translated into some other format. When the user selects an application requiring translation and clicks the Open button, Macintosh Easy Open calls the appropriate translation extension to translate the original document. During the translation, Macintosh Easy Open displays a translation progress dialog box, as shown in Figure 7-4.

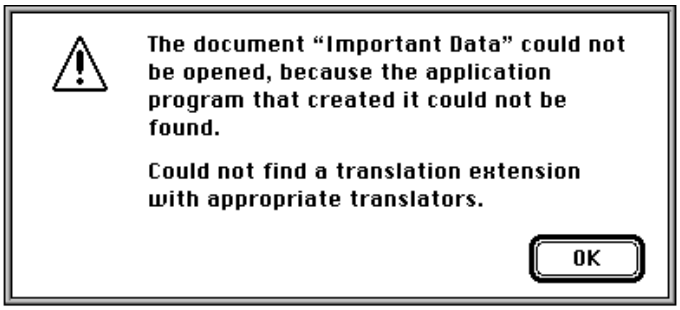
Figure 7-4 A translation progress dialog box



The progress dialog box displays the name of the document being translated, its original format, and its target format. The top portion of the dialog box shows an advertisement provided by the particular translation extension that Macintosh Easy Open called to perform the translation. (In this case, the Hang Ten Translation Extension is being used.) It's possible that two or more translation extensions can translate the same original document; if so, they'll all be listed in the translation choices dialog box.

If none of the available translation extensions can translate a particular document, the Finder may present a modified version of the application-unavailable alert box, shown in Figure 7-5.

Figure 7-5 The modified application-unavailable alert box

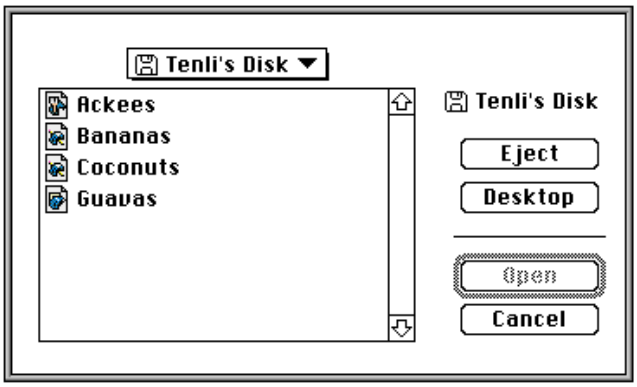


To have another application open a document, the user can drop-launch the document. (To **drop-launch** a document is to drag the document's icon onto the application's icon.) If Macintosh Easy Open knows how to translate the document into a format that can be opened by that application, the Finder highlights the application's icon as the user drags the document icon over it. When the user drop-launches the document, Macintosh Easy Open displays a dialog box that is similar to the translation choices dialog box (see Figure 7-3).

Opening Documents Within an Application

When present in the operating environment, Macintosh Easy Open modifies the Standard File Package so that its file-opening dialog boxes display not only the file types your application can open by itself but also the file types that can be translated into those your application can open. The result is that users can open more documents using your application than they previously could. Figure 7-6 shows the enhanced file-opening dialog box.

Figure 7-6 The enhanced file-opening dialog box



In the case shown in Figure 7-6, the application can open SurfWriter documents without translating them. In addition, Macintosh Easy Open can translate SurfDB and SurfPaint documents to SurfWriter documents; as a result, any SurfDB and SurfPaint documents in the current directory are displayed in the dialog box.

If the user selects a document that your application can open only after some sort of translation, Macintosh Easy Open displays the translation progress dialog box (shown in Figure 7-4) and translates the document into a format that your application recognizes.

Notice in Figure 7-6 that the small, black-and-white generic document icons (of type `'SICN'`) usually displayed by the Standard File Package have been replaced by small color icons (displayed in this figure in grayscale) that are specific to each type of document. When Macintosh Easy Open is present, the Standard File Package uses small color icons (of type `'ics4'` or `'ics8'`, according to the current bit depth of the display device) to show document types. This allows the user to distinguish more easily between documents of different file types and provides a clue to which documents belong to your application and which belong to some other application but can be opened after translation.

IMPORTANT

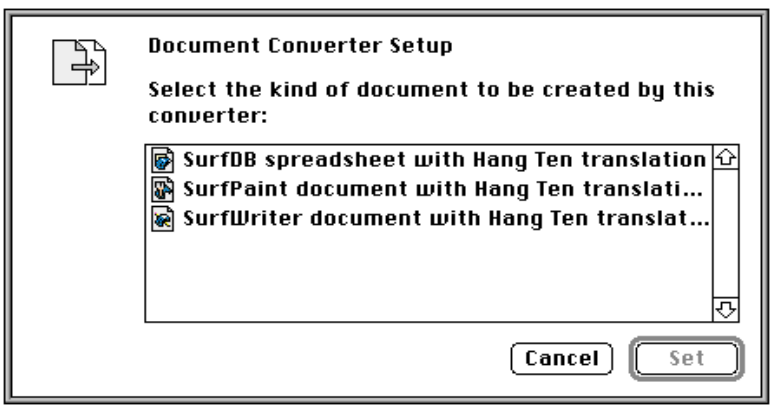
To have the Standard File Package display your application's small color icons in the file-opening dialog box, your application's resource fork should contain the appropriate small color icons (of type `'ics4'` or `'ics8'`). In addition, if your application uses custom Standard File Package file-opening dialog boxes, your resource fork should contain a dialog color table resource (of type `'dctb'`) whose resource ID is the same as the resource ID of the dialog box. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information about small color icons; see the chapter "Dialog Manager" in that same book for information about dialog color tables. ▲

Translating Documents on the Desktop

Macintosh Easy Open includes a tool, called Document Converter, that allows users to convert documents without opening them. This tool is useful if a user wants to convert a number of documents (batch translation) or wants to give the translated documents to other users who don't have either Macintosh Easy Open or the appropriate translation extensions installed on their machines.

To translate documents on the desktop, the user needs to configure the Document Converter tool. When the user opens the Document Converter, it displays the dialog box shown in Figure 7-7.

Figure 7-7 Document Converter configuration dialog box



This dialog box lists target document types, not applications. The user selects a target document type and clicks the Set button to complete the configuration. At that point, the Document Converter application quits and changes its own name to reflect the conversion path of documents subsequently dropped onto it.

Once the Document Converter has been configured, the user can translate documents by dropping them onto the Document Converter icon. The Document Converter creates a new document in the target format and leaves the original document unmodified. The user can also drop a group of documents—or even a folder of documents—onto the Document Converter icon; in these cases, the Document Convertor translates all the documents in the group.

Sharing Data Between Applications

Macintosh Easy Open can translate not only documents (data stored in files) but also scraps (data stored in memory) and other data. For instance, when a user copies a selection in one document and pastes the data into a document created by some other application, Macintosh Easy Open steps in, if necessary, to translate the data from its original format (as contained on the Clipboard) to the format of the target document. Because the source and target formats are known, Macintosh Easy Open doesn't need to present the translation choices dialog box (shown in Figure 7-3 on page 7-6). Instead, Macintosh Easy Open proceeds directly with the translation. The only sign that Macintosh Easy Open is at work is the translation progress dialog box.

Data shared in editions might also need to be translated from one format to another. When a user subscribes to an edition (or updates an existing subscriber) and the data in the edition is not already in the format of the subscribing application, Macintosh Easy Open translates the data. Once again, it displays the translation progress dialog box to show the user that it's at work.

Using the Translation Manager

Most applications benefit from the services of Macintosh Easy Open automatically if they use the standard Macintosh system software (such as the Standard File Package, the Edition Manager, and the Scrap Manager) to open files or exchange data with other applications. If the appropriate translators are present on a particular computer, Macintosh Easy Open implicitly translates file and scrap formats into those supported by your application. To facilitate this translation, however, you should

- Make your application stationery-aware. When Macintosh Easy Open passes your application a translated document, the document's stationery bit is set if your application is stationery-aware. The user should be prompted to save any changes to the translated document under a new name when closing the document.
- Add a resource of type `'open'` to your application. This resource indicates what file types your application can open. See "Declaring the File Types Your Application Can Open" on page 7-13 for complete details.

- Add a resource of type `'kind'` to your application. This resource allows the Finder to display custom kind strings in its windows. See "Declaring Custom Kind Strings" beginning on page 7-14 for complete details.
- Add a resource of type `'dctb'` to your application if it uses custom Standard File Package file-opening dialog boxes. This resource allows the Standard File Package to display the enhanced small color icons in its dialog boxes. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on creating resources of type `'dctb'`.
- Avoid using a file filter function as the only method of specifying file types when calling the Standard File Package routines `StandardGetFile` and `CustomGetFile` (or the original `SFGetFile` and `SFPGetFile`). Instead of a file filter function (or in addition to it), you should use the `typeList` parameter to specify file types to list in the file-opening dialog box. Alternatively, you can pass the special value `kUseOpenResourceTypes` in the `numTypes` parameter to have the file types read from your application's `'open'` resource. See "Using File-Opening Dialog Boxes" on page 7-15 for more details.
- Use the Scrap Manager properly:
  - Put formats on the scrap in order of fidelity.
  - Get formats from the scrap in the order that your application can best interpret.
  - Don't call `GetScrap` unless the user has just pasted, because doing so may cause a lengthy translation.
  - Be able to put the popular scrap formats (such as `'styl'`) on the scrap.
  - Don't rely on the `offset` parameter returned by `GetScrap`. It is undefined after implicit translation.
- Avoid using `'TEXT'` as a file type of a document your application creates unless the document contains plain ASCII text intended to be viewed by the user as plain text.
- Use file types that accurately indicate the format type of the documents your application creates. When you revise your application and make extensive changes to the file format of a document, previous versions of your application will not be able to read the document. In this case, you should assign a different file type to the new format.

If your application does not use the Standard File Package to allow the user to select files to open, you can use the Translation Manager to make your application compatible with Macintosh Easy Open. See "Translating Files Explicitly" on page 7-17 for details.

Checking for the Translation Manager

Macintosh Easy Open and the Translation Manager are not available in all system software versions. You can use the `Gestalt` function to determine whether the services you need are available before calling them. To get information about the Translation Manager, you pass `Gestalt` the selector `gestaltTranslationAttr`.

```
CONST
   gestaltTranslationAttr        = 'xlat';   {Translation Manager}
```

`Gestalt` returns in the `response` parameter a bit field that encodes information about the Translation Manager. Currently only 1 bit is used:

```
CONST
   gestaltTranslationMgrExists   = 0;        {TM is present}
```

If the indicated bit (bit 0) is set, the Translation Manager is available and you can safely call the routines it provides. Otherwise, if that bit is clear, the Translation Manager is not available.

As you have seen, Macintosh Easy Open works with the Standard File Package, the Edition Manager, and the Scrap Manager to translate files and scraps implicitly. In most cases, you don't need to know that a file or scrap has been implicitly translated, but in some cases you might need this information. You can use `Gestalt` to determine whether these other system software parts are capable of supporting the capabilities of Macintosh Easy Open. Listing 7-1 lists the translation-specific Gestalt selectors and bit numbers of the `response` parameter for the Standard File Package, the Edition Manager, and the Scrap Manager.

Listing 7-1 Translation-specific selectors and response bit for `Gestalt`

```
CONST
   gestaltStandardFileAttr                = 'stdf';   {Standard File Package}
   gestaltStandardFileTranslationAware    = 1;
   gestaltStandardFileHasColorIcons       = 2;

   gestaltEditionMgrAttr                  = 'edtn';   {Edition Manager}
   gestaltEditionMgrTranslationAware      = 1;

   gestaltScrapMgrAttr                    = 'scra';   {Scrap Manager}
   gestaltScrapMgrTranslationAware        = 0;
```

For complete information about the `Gestalt` function, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

Declaring the File Types Your Application Can Open

In system software versions 7.0 and later, the Finder determines which types of files your application can open by inspecting the resources of type `'FREF'` whose resource IDs are listed in your application's bundle (that is, your application's resource of type `'BNDL'`). The Finder uses this information to determine which file types can be drop-launched on your application. All file types in the `'FREF'` resources listed in your application's bundle, regardless of whether they have associated icons, are considered droppable on your application.

Note
See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a complete description of resources of types `'FREF'` and `'BNDL'`. ◆

In some cases, however, your application might include `'FREF'` resources for file types that you don't want the user to open. For example, your application might use non-document files such as dictionaries and help files. Even though these files should have icons and hence deserve `'FREF'` resources, their contents should not be displayed to the user. Similarly, your application might read data from preferences files; this data is intended to be used internally by the application, not opened by the user as a document.

Because the list of file types your application can open may be different from the list of types that have icons, the Translation Manager defines a new resource of type `'open'`. The **open resource** declares which file types your application can open as documents (and hence can be dropped onto your application). Listing 7-2 shows a sample resource of type `'open'`, in Rez input format.

Listing 7-2 A sample resource of type `'open'`

```
/*open resource for TeachText*/
resource 'open' (128)
{
   'ttxt', { 'ttro', 'PICT', 'TEXT' }
};
```

An open resource consists of an application signature followed by a list of file types. It indicates that the specified application can open files whose types occur in the list. For example, TeachText can open documents created in its own private format, `'ttro'`, as well as documents of file type `'PICT'` and `'TEXT'`. If Macintosh Easy Open is available, the Finder allows the user to drop documents of those types onto the application. In addition, if any translation extensions are installed, all documents that can be translated to one of the specified types can also be dropped on the application. So, if a translation extension exists that can translate documents from type `'SURF'` to type `'ttro'`, the user can drop SurfWriter documents onto TeachText.

You should list file types in your open resource in order of decreasing preference. If the Translation Manager has to choose between multiple file types as the destination file type for a translation, it chooses the file type that occurs earliest in the list.

The open resource is also used by the routine `StandardOpenDialog` to determine which documents should be listed in the file-opening dialog box. See "Using File-Opening Dialog Boxes" on page 7-15 for details on `StandardOpenDialog`.

IMPORTANT

If you use the `StandardOpenDialog` function, the open resource in your application should have resource ID 128. ▲

Your application might need to determine dynamically which types of files it can open (perhaps by inspecting which filters are available in a certain folder). If so, you cannot list those file types statically in an open resource. Instead, you can write a simple translation extension to generate a list of openable file types at runtime. See "Writing Application Translation Extensions" beginning on page 7-35 for details.

Declaring Custom Kind Strings

A file's **kind string** is the string displayed in the "Kind" column in a Finder window when a folder's contents are viewed by name, size, kind, label, or date (that is, by any method other than by icon or small icon). The Finder determines the kind string for a file by taking the name of the application that created it and, in the case of English, appending "document" to that name (for example "SurfWriter document"). If the user does not have the application that created the file, the kind string is simply "document".

Note

Localized versions of the Finder determine the kind string in other ways. For instance, the Finder may prepend some string (for example, "document de SurfWriter"). ◆

If the application isn't available on the computer (the situation in which the user is most likely to want the kind information), the kind string is not particularly helpful. In that case, the displayed string is "document" (or some localization thereof), and the user has no idea which application created it. Moreover, the documents of applications that support many kinds of documents all have the same kind string, even though those documents may be of entirely different kinds (such as word-processing documents, spreadsheet documents, graphics documents, and so forth). It would be better to have the Finder list more information about a document than its creator.

To solve these problems, Macintosh Easy Open allows you to define a custom kind string for each type of file your application creates. You do this by including a **kind resource** (a resource of type `'kind'`) in your application's resource file. The custom kind strings defined in a kind resource override the algorithm the Finder uses to create kind strings. Listing 7-3 shows a sample kind resource, in Rez input format.

Listing 7-3 A sample resource of type `'kind'`

```
/*sample kind resource for SurfSoft Works*/
resource 'kind' (1000)
{
   'WAVE',
   verUS,
   {
      ftApplicationName,      "SurfSoft Works",
      'SWTD',                 "SurfSoft Works text document",
      'SWSS',                 "SurfSoft Works spreadsheet",
      'SWDB',                 “SurfSoft Works database",
   }
};
```

A kind resource consists of an application signature, a localization code, and a list of file types and their corresponding kind strings. Each file type is associated with one kind string.

To reduce the number of entries in a kind resource, you can declare your application's name by including an entry having the special file type `ftApplicationName`, as illustrated in Listing 7-3. Then, whenever Macintosh Easy Open encounters a document that belongs to your application but whose file type isn't listed in your application's kind resource, the Finder uses its standard algorithm to generate a kind string in the form "<application name> document".

Note

Because a kind resource contains the application signature, an application's kind resource can be located in some file other than the application's resource file. This feature allows translation extensions to provide kind strings for applications that might not be present on a particular computer. However, the kind resource in an application's resource file overrides any kind resource located elsewhere. ◆

The Finder uses only custom kind strings that have the same localization as the current system itself.

Using File-Opening Dialog Boxes

Macintosh Easy Open works with the Standard File Package to list in the file-opening dialog box all files that your application can open, including those that it can open after they have been translated from their current format to some new format. In general, you don't need to rewrite your application (or even include any additional resources) to receive this service. Macintosh Easy Open provides it automatically when present in the operating environment.

There are, however, some cases in which Macintosh Easy Open cannot provide this service and that might therefore require you to modify your application if you want to maximize compatibility with Macintosh Easy Open. In particular, if you use a file filter function when calling the Standard File Package routines as the *only* way of determining which files appear in the list of files to open, Macintosh Easy Open cannot safely add any files to that list. This is a problem only when you specify –1 as the value for the `numTypes` parameter in a call to `StandardGetFile` or `CustomGetFile`.

Note

For complete information about file filter functions, see the chapter "Standard File Package" in *Inside Macintosh: Files*. ◆

If you use a file filter function when calling the Standard File Package, you should make sure that the list of file types you pass in the `typeList` parameter isn't empty (that is, that the value of the `numTypes` parameter isn't 0). In that case, Macintosh Easy Open is able to expand the list of file types your application can open, regardless of whether you use a file filter function. Macintosh Easy Open inspects the file types passed in the `typeList` parameter and adds to them all file types that can be translated into those file types. In short, you can use a file filter function and benefit from the translation services of Macintosh Easy Open if you specify a non-empty list of file types in the `typeList` parameter.

IMPORTANT

If for some reason you want to prevent Macintosh Easy Open from expanding the list of file types your application can open, simply set the `numTypes` parameter to –1 when calling `StandardGetFile` or `CustomGetFile`. ▲

When Macintosh Easy Open is present, you can pass the special value `kUseOpenResourceTypes` in the `numTypes` parameter to have the file types read from your application's `'open'` resource.

```
CONST
   kUseOpenResourceTypes      = -2;
```

When `numTypes` is set to `kUseOpenResourceTypes`, `typeList` is set to `NIL`, and `fileFilter` is set to `NIL`, the Standard File Package displays in the file-opening dialog box all files whose types are listed in your application's `'open'` resource (having resource ID 128) as well as all files whose types can be translated into those types.

You can achieve this same result by calling the new Standard File Package function `StandardOpenDialog`.

```
FUNCTION StandardOpenDialog (VAR reply: StandardFileReply): OSErr;
```

The `StandardOpenDialog` function operates exactly like the `StandardGetFile` function, whose parameters `fileFilter`, `numTypes`, and `typeList` are given the values `NIL`, `kUseOpenResourceTypes`, and `NIL`, respectively.

IMPORTANT

The `StandardOpenDialog` function is implemented as glue code and is available in System 6 and later if you link your application with the appropriate object library. ▲

Translating Files Explicitly

It's possible that your application might open some document files without the assistance of the Finder or the Standard File Package. For example, your application might execute a script that contains the name of a file to open. Because you're bypassing the system software services that invoke implicit translation, you might need to modify your application to perform **explicit translation** (the conversion of a file or scrap with direct intervention from your application). The Translation Manager provides several routines that you can use to retrieve information about documents and about the document types that an application can open, as well as to translate documents from one format to another.

IMPORTANT

Before calling the routines described in this section, you must make sure that they are available in the current operating environment. See "Checking for the Translation Manager" on page 7-12 for details. ▲

You can use the `GetFileTypesThatAppCanNativelyOpen` function to get a list of file types that an application can open by itself. This function takes a volume reference number (where the application resides), an application signature, and a pointer to a buffer to be filled with up to 64 file types. It returns a pointer to the list of the file types that the application can open without translation.

You can use the `ExtendFileTypeList` function to get a list of all file types that the Translation Manager can translate into file types in a given list. This routine takes the original list, the number of file types in it, a pointer to a buffer to be filled with file types, and the maximum number of file types that can be put into the extended list. The `ExtendFileTypeList` function returns a list of all the file types that can be translated into some type in the original list.

You can use the `CanDocBeOpened` function to verify that a specified application can open the document that it is being requested to open. It takes the source document record, the volume reference number of the application that is to open the document, the creator application signature, and the list of file types that the application can open without translation. It returns a document-opening method (`howToOpen`) and document-translation method (`howToTranslate`). The choices for document-opening method are

- `domCannot`
- `domNative`
- `domTranslateFirst`
- `domWildcard`

The Translation Manager uses `howToTranslate` to get information on converting the document into a format the application can read. For more information on the `CanDocBeOpened` function, see page 7-40.

You can call the function `TranslateFile` to translate a file from one format to another. It takes the source document record, the destination document record, and the `howToTranslate` parameter returned by `CanDocBeOpened`. In the destination document record, `TranslateFile` returns the name and location of the translated file.

Writing a Translation Extension

A translation extension is a component that works with Macintosh Easy Open to provide data recognition and translation capabilities. Because a translation extension is a component, it must be able to respond to the required request codes sent by the Component Manager. In addition, a translation extension can

- communicate its translation capability to Macintosh Easy Open
- identify the formats of specific documents and scraps
- translate documents and scraps

Translation extensions can identify and translate files, scraps, or both. You specify whether a translation extension handles files or scraps by setting bits in the component flags field in the component resource (see "Creating a Translation Extension" beginning on page 7-19 for details).

IMPORTANT

The information in this section describes how to write translation extensions. If you simply want to make your application compatible with Macintosh Easy Open, see "Using the Translation Manager" beginning on page 7-10. If your application needs to determine dynamically which file types it can open, see "Writing Application Translation Extensions" beginning on page 7-35. ▲

Macintosh Easy Open and the Translation Manager specify file and scrap formats using the `FileType` and `ScrapType` data types:

```
TYPE
   FileType                = OSType;   {file types}
   ScrapType               = ResType;  {scrap types}
```

The `ScrapType` data type describes the format of data in memory. In general, the scrap types used by Macintosh Easy Open are identical to scrap types used by the Scrap Manager. There is, however, one notable exception. Macintosh Easy Open defines a new scrap type, `'stxt'`, to describe styled text. A scrap having format `'stxt'` is formed by appending the text (as contained in a scrap of format `'TEXT'`) to the style information (as contained in a scrap of format `'styl'`). This is necessary to have a single scrap to pass to your scrap translation extension.

The `FileType` data type describes the format of a file. Often, but not always, the format of a file's data can be identified by inspecting the file's type, as maintained in the hierarchical file system catalog file (hereafter called the file's **catalog type**). For purposes of translation, however, it is sometimes necessary to use a more specific identification. For example, some developer might revise an application but retain the existing file type for documents the application creates. This could cause problems for translation extensions, which might be able to translate a specific version of the application's data format but not later ones. Similarly, some applications that create files on Macintosh computers (such as electronic mail programs or disk-mounting utilities) often use standard file types (such as `'TEXT'` or `'BINA'`) as the default new file type. Once again, your translation extension needs more information about the actual format of the data in the file before it can translate it to some other format.

To avoid problems with using a file's catalog type as the only indication of the file's data format, Macintosh Easy Open and the Translation Manager allow you to define a **translation file type.** As just indicated, the catalog file type is often sufficient as the translation file type. However, Macintosh Easy Open always gives your translation extension the opportunity to inspect a particular file to see whether its catalog file type is in fact sufficient for translation purposes. If your extension can identify a more specific format, it should return that information to Macintosh Easy Open. (Ideally, application developers should assign catalog file types that can be used as translation file types.)

The rest of this section describes how to create a file translation extension. You create a scrap translation extension in like fashion, substituting the scrap data types for the corresponding file data types.

Creating a Translation Extension

A translation extension is a component. It contains a number of resources, including icons, strings, pictures, and the standard component resource (a resource of type `'thng'`) required of any Component Manager component. In addition, a translation extension must contain code to handle required request codes passed to it by the Component Manager as well as translation-specific request codes.

For complete details on components and their structure, see the chapter "Component Manager" in this book. This section provides specific information about translation extensions.

The component resource binds together all the relevant resources contained in a component; its structure is defined by the `ComponentResource` data type.

```
TYPE ComponentResource =
   RECORD
      cd:               ComponentDescription;
      component:        ResourceSpec;
      componentName:    ResourceSpec;
      componentInfo:    ResourceSpec;
      componentIcon:    ResourceSpec;
   END;
```

The `component` field specifies the resource type and resource ID of the component's executable code. By convention, for translation extensions this resource should be of type `'xlat'`. (You can, however, specify some other resource type if you wish.) The resource ID can be any integer greater than or equal to 128. See the following section for further information about this code resource. The `ResourceSpec` data type has this structure:

```
TYPE ResourceSpec =
   RECORD
      resourceType:     ResType;
      resourceID:       Integer;
   END;
```

The `componentName` field specifies the resource type and resource ID of the resource that contains the component's name. Usually the name is contained in a resource of type `'STR '`. Macintosh Easy Open uses the component's name in several of the dialog boxes it displays. (For example, in Figure 7-3 on page 7-6, one of the translation extensions has the component name "Hang Ten.") This string should be as short as possible.

The `componentInfo` field specifies the resource type and resource ID of the resource that contains a description of the component. Usually the description is contained in a resource of type `'STR '`. This information is not currently used by Macintosh Easy Open, but some development tools may use it.

The `componentIcon` field specifies the resource type and resource ID of the resource that contains an icon for the component. Usually the icon is contained in a resource of type `'ICON'`. This icon is not currently used by Macintosh Easy Open, but some development tools may use it.

Note
The icon displayed in Figure 7-4 on page 7-7 is part of the translation extension's advertisement; it is not supplied by Macintosh Easy Open itself. ◆

The `cd` field of the `ComponentResource` structure is a component description record, which contains additional information about the component. A component description record is defined by the `ComponentDescription` data structure.

```
TYPE ComponentDescription =
   RECORD
      componentType:          LongInt;
      componentSubType:       LongInt;
      componentManufacturer:  LongInt;
      componentFlags:         LongInt;
      componentFlagsMask:     LongInt;
   END;
```

For translation extensions, the `componentType` field must be set to `'xlat'`. In addition, the `componentSubType` field must be set to 0 (because there are currently no subtypes of translation extensions). The `componentManufacturer` field identifies the supplier of the component. You should register your component with Apple's Component Registry Group to receive a unique manufacturer code; this code typically corresponds to the signature of your translation extension.

The `componentFlags` field of the component description for a translation extension contains bit flags that encode information about the extension. Currently, you can use this field to specify whether the extension supports file translation routines or scrap translation routines, or both. (See the chapter "Component Manager" in this book for information about the standard flags that you can also specify in the `componentFlags` field.)

```
CONST
   kSupportsFileTranslation   = 1;  {file translation extension}
   kSupportsScrapTranslation  = 2;  {scrap translation extension}
```

You should set the `componentFlagsMask` field to 0.

IMPORTANT
For compatibility with early versions of the Component Manager, a `'thng'` resource should be locked. You can set the other resource attributes in any way you wish. ▲

In addition to the component resource, a translation extension must contain the string and icon resources specified in the component resource (for example, the resource that contains the extension's name). You might also want to include several other resources in the translation extension, including the standard `'BNDL'`, `'FREF'`, and `'ICN#'` resources used by the Finder and a `'PICT'` resource that contains an advertisement or banner to be displayed in the translation progress dialog box. You should also include a `'kind'` resource listing kind strings for all the file types your extension can translate from or to; this allows the Finder to display correct kind strings once your extension is installed. Listing 7-4 shows, in Rez input format, the component resource and associated resources of a sample translation extension.

Listing 7-4 Sample resources for a translation extension

```
/*a component resource*/
resource 'thng' (128, locked) {
   'xlat',                    /*all translation extensions have this type*/
   0,                         /*subtype is unused*/
   'MYCO',                    /*creator signature of extension*/
   kSupportsFileTranslation,  /*only file routines are implemented*/
   0,                         /*mask is unused and should be 0*/
   'xlat',128,                /*resource type & ID of translation extension*/
   'STR ',128,                /*resource type and ID of name string*/
   'STR ',129,                /*resource type and ID of information string*/
   'ICON',128                 /*resource type and ID of icon*/
};

/*strings*/
resource 'STR ' (128, purgeable) {
   "Hang Ten"
};

resource 'STR ' (129, purgeable) {
   "Hang Ten Translation Extension"
};

/*an icon*/
resource 'ICON' (128, purgeable) {
   $"7FFF FFF0 8000 0008 8000 0008 8000 0008"
   $"8000 0008 8000 0008 8000 0008 8000 0008"
   $"A000 0008 D000 000A 9000 000D 1000 0009"
   $"1000 0001 1000 0001 1000 0001 1000 0001"
   $"1000 0001 1000 0001 1000 0001 1000 0001"
   $"1000 0009 9000 000D D000 000A A000 0008"
   $"8000 0008 8000 0008 8000 0008 8000 0008"
```

```
   $"8000 0008 8000 0008 8000 0008 7FFF FFF0",
};

/*kind strings for document types supported by this extension*/
resource 'kind' (128, purgeable) {
   'SURF',
   verUS,
   {
      ftApplicationName,      "SurfWriter",
      'SURF',                 "SurfWriter document",
   }
};

resource 'kind' (129, purgeable) {
   'SPNT',
   verUS,
   {
      ftApplicationName ,     "SurfPaint",
      'SPNT',                 "SurfPaint document",
   }
};

resource 'kind' (130, purgeable) {
   'ttxt',
   verUS,
   {
      ftApplicationName ,     "TeachText",
      'ttro',                 "TeachText document",
   }
};
```

Your translation extension is contained in a resource file. The creator of the file can be any type you wish, but the type of the file must be `'thng'`. If the extension contains a `'BNDL'` resource, then the file's bundle bit must be set.

IMPORTANT

The Finder looks for open and kind resources only in files that have their bundle bit set. ▲

Dispatching to Translation Extension-Defined Routines

As explained in the previous section, the code stored in the translation extension component should be contained in a resource of type `'xlat'`. The Component Manager expects that the entry point in this resource is a function having this format:

```
FUNCTION TranslateEntry (VAR params: ComponentParameters;
                         storage: Handle): ComponentResult;
```

The Component Manager calls your extension by passing `TranslateEntry` a request code in the `params.what` field of the components parameter record passed in the `params` parameter; `TranslateEntry` must interpret the request code and possibly dispatch to some other routine in the resource. Your extension must be able to handle the required request codes, defined by these constants:

```
CONST
   kComponentOpenSelect                 = -1;
   kComponentCloseSelect                = -2;
   kComponentCanDoSelect                = -3;
   kComponentVersionSelect              = -4;
```

For complete details on required request codes, see the chapter "Component Manager" in this book.

In addition, your extension must be able to respond to translation-specific request codes. Currently, Macintosh Easy Open defines these six request codes:

```
CONST
   kTranslateGetFileTranslationList     = 0;
   kTranslateIdentifyFile               = 1;
   kTranslateTranslateFile              = 2;
   kTranslateGetScrapTranslationList    = 10;
   kTranslateIdentifyScrap              = 11;
   kTranslateTranslateScrap             = 12;
```

You can respond to these request codes by calling the Component Manager routine `CallComponentFunctionWithStorage`, passing it a pointer to a translation extension-defined routine. Listing 7-5 illustrates how to define a file translation extension entry point routine.

Listing 7-5 Handling Component Manager request codes

```
FUNCTION TranslateEntry (VAR params: ComponentParameters;
                         storage: Handle): ComponentResult;
TYPE
   LongPtr     =  ^LongInt;
   LongHandle  =  ^LongPtr;
VAR
   mySelf:        ComponentInstance;
   myHandle:      Handle;
   selector:      Integer;
BEGIN
   CASE params.what OF
      kComponentOpenSelect:                {component is opening}
         BEGIN
            mySelf := ComponentInstance(params.params[0]);
            myHandle := NewHandle(SizeOf(ComponentInstance));
            IF myHandle <> NIL THEN
            BEGIN
               LongHandle(myHandle)^^ := ORD4(mySelf);
               SetComponentInstanceStorage(mySelf, myHandle);
               TranslateEntry := noErr;
            END
            ELSE
               TranslateEntry := MemError;
         END;
      kComponentCloseSelect:               {component is closing; clean up}
         BEGIN
            IF storage <> NIL THEN
               DisposeHandle(storage);
            TranslateEntry := noErr;
         END;
      kComponentCanDoSelect:               {return known selectors}
         BEGIN
            selector := Integer((Ptr(params.params)^));
            IF (((kComponentVersionSelect <= selector)
                  AND (selector <= kComponentOpenSelect))
               OR ((kTranslateGetFileTranslationList <= selector)
                  AND (selector <= kTranslateTranslateFile))) THEN
               TranslateEntry := 1
            ELSE
               TranslateEntry := 0;
         END;
```

```
      kComponentVersionSelect:               {provide version number}
         TranslateEntry := kMyTranslateVersionNumber;
      kTranslateGetFileTranslationList:      {give file translation list}
         TranslateEntry := CallComponentFunctionWithStorage
                           (Handle(storage^^), params,
                           ComponentFunction(@DoGetFileTranslationList));

      kTranslateIdentifyFile:                {identify a file}
         TranslateEntry := CallComponentFunctionWithStorage
                           (Handle(storage^^), params,
                           ComponentFunction(@DoIdentifyFile));
      kTranslateTranslateFile:               {translate a file}
         TranslateEntry := CallComponentFunctionWithStorage
                           (Handle(storage^^), params,
                           ComponentFunction(@DoTranslateFile));
      OTHERWISE                              {unrecognized selector}
         TranslateEntry := badComponentSelector;
   END; {CASE}
END;
```

As you can see, the `TranslateEntry` function defined in Listing 7-5 simply inspects the `params.what` field to determine which request code to handle. For translation-specific request codes, it dispatches to the appropriate function in the translation extension. See the following three sections for more details on handling translation-specific request codes.

Your extension can be dynamically loaded or unloaded at any time. When Macintosh Easy Open first discovers the extension, it loads it into memory and then asks it to return a list specifying which file or scrap types it can translate into which other types. Your extension is also called during a translation to identify files or scraps and, if necessary, to translate them.

Macintosh Easy Open loads your extension into a subheap of some existing heap. In all likelihood, your extension is loaded into either the system heap or temporary memory. In some cases, however, your extension might be loaded into an application's heap. Your extension is guaranteed 32 KB of available heap space. You should do all allocation in that heap using normal Memory Manager routines. Any memory leaks are reclaimed when your routine returns and the heap is destroyed.

There is no support for using global variables in the dispatcher defined in Listing 7-5. In general, the routines you need to implement are separate and self-contained, and so you shouldn't need to use global variables. You can, however, have your dispatcher set up an A5 world that contains global variables.

▲ **WARNING**
If you use PC-relative global variables (that is, data addressed relative to the program counter), be warned that the Component Manager may purge and reload your code resource. Therefore, all PC-relative global variables must be preinitialized at compile time (not at load time). ▲

If you need to access resources that are stored in your translation extension, you should use `OpenComponentResFile` and `CloseComponentResFile`. The open routine requires the `ComponentInstance` parameter supplied to your routine. See Listing 7-7 on page 7-33 for an example. You should not call the Resource Manager routines such as `OpenResFile` or `CloseResFile`.

▲ **WARNING**
Do not leave any resource files open when your translation extension exits. Their maps will be left in the subheap when the subheap is freed, causing the Resource Manager to crash. ▲

The following sections show how to respond to the `kTranslateGetFileTranslationList`, `kTranslateIdentifyFile`, and `kTranslateTranslateFile` request codes by defining the three file translation extension functions `DoGetFileTranslationList`, `DoIdentifyFile`, and `DoTranslateFile`. You would handle scrap translation in a similar manner.

Creating a Translation List

Your translation extension must be able to inform Macintosh Easy Open of its translation capabilities in response to the `kTranslateGetFileTranslationList` request code. To do this, you can define a `DoGetFileTranslationList` function in which you fill in a **file translation list,** defined by a `FileTranslationList` record. From the file translation list you return, Macintosh Easy Open learns which types of files your extension can translate into which other types. On the basis of this information, it may later call your extension to identify a particular document and possibly to translate it.

The `FileTranslationList` record has this structure:

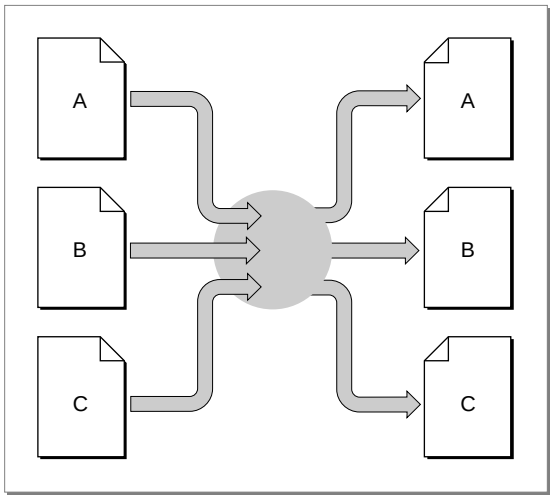
```
TYPE FileTranslationList =
   RECORD
      modDate:                LongInt;
      groupCount:             LongInt;
      {group1SrcCount:        LongInt;}
      {group1SrcEntrySize:    LongInt;}
      {group1SrcTypes:        ARRAY[1..group1SrcCount] OF FileTypeSpec;}
      {group1DstCount:        LongInt;}
      {group1DstEntrySize:    LongInt;}
      {group1DstTypes:        ARRAY[1..group1DstCount] OF FileTypeSpec;}
      {repeat above six lines for a total of groupCount times}
   END;
```

This record contains a modification date and a count of the number of translation groups that follow. Each **translation group** in the file translation list specifies a collection of file types from which the extension can translate (the `group1SrcTypes` field) and a collection of file types into which the extension can translate (the `group1DstTypes` field). Within a translation group, your extension must be able to translate any of the source types into any of the destination types.

You might have different translation groups corresponding to different categories of documents. For instance, you can place word-processing documents in one group, spreadsheet documents in another, and so on. You are, however, free to group file types in whatever manner you like.

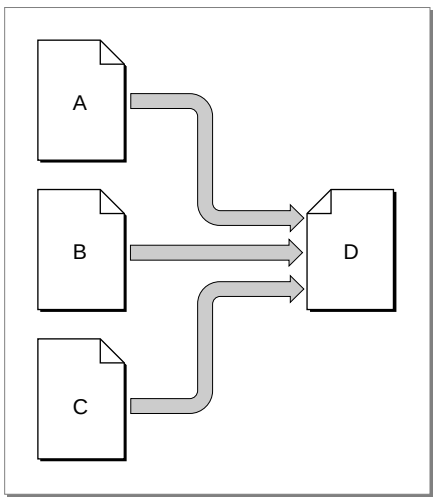
In most cases, `group1SrcCount` and `group1DstCount` will each be greater than 1, because most translators operate by translating through a particular data model. In these cases, it's also quite likely that the source and destination file types overlap or even coincide. Figure 7-8 illustrates a typical translation group.

Figure 7-8 A translation group with multiple source and destination types



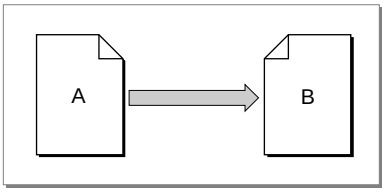
Similarly, you might write a translation extension that converts other file types into your own proprietary document format. In this case, you would have multiple source document types but only one destination type (`group1DstCount` equal to 1), as illustrated in Figure 7-9.

Figure 7-9 A translation group with a single destination type



It's possible, however, to have both `group1SrcCount` and `group1DstCount` equal to 1. This kind of translation is known as **point-to-point translation.** Figure 7-10 illustrates point-to-point translation.

Figure 7-10 Point-to-point translation



Note
The number of translation groups you can specify in a file translation list is limited by memory considerations only. ◆

Within any particular group of file types, you specify a particular document format using a **file type specification,** defined by the `FileTypeSpec` data type.

```
TYPE FileTypeSpec =
   RECORD
      format:           FileType;
      hint:             LongInt;
      flags:            TranslationAttributes;
      catInfoType:      OSType;
      catInfoCreator:   OSType;
   END;
```

A file type specification includes the file type, a hint reserved for use by your extension, a flags field, and the original file type and creator. See "File Type Specifications" beginning on page 7-46 for complete details on these fields.

Listing 7-6 shows a simple routine that creates a file translation list. The translation extension containing this routine can translate both SurfWriter and SurfPaint documents to a format understood by TeachText.

Listing 7-6 Creating a file translation list

```
FUNCTION DoGetFileTranslationList
                    (self: ComponentInstance;
                     translationList: FileTranslationListHandle)
                     : ComponentResult;
```

```
TYPE
   MyList =
      RECORD
         modDate:                LongInt;
         groupCount:             LongInt;
         group1SrcCount:         LongInt;
         group1SrcEntrySize:     LongInt;
         group1SrcTypes:         ARRAY[1..2] OF FileTypeSpec;
         group1DstCount:         LongInt;
         group1DstEntrySize:     LongInt;
         group1DstTypes:         ARRAY[1..1] OF FileTypeSpec;
      END;
   MyListPtr      = ^MyList;
   MyListHandle   = ^MyListPtr;
VAR
   myErr:      OSErr;
   myPtr:      MyListPtr;
CONST
   kStamp      = $A74520A8;      {date of original list creation}
BEGIN
   myErr := noErr;
   IF translationList^^.modDate <> kStamp THEN
   BEGIN
      {resize the handle so there's enough room}
      SetHandleSize(Handle(translationList), SizeOf(MyList));
      myErr := MemError;
      IF myErr = noErr THEN
      BEGIN
         myPtr := MyListHandle(translationList)^;
         WITH myPtr^ DO
         BEGIN
            modDate := kStamp;              {set creation date}
            groupCount := 1;                {only 1 translation group}
            group1SrcCount := 2;            {source side has two types}
            group1SrcEntrySize := SizeOf(FileTypeSpec);
            WITH group1SrcTypes[1] DO
            BEGIN
               format := 'SURF';            {SurfWriter document format}
               hint := 0;                   {no hint}
               flags := 0;                  {no flags}
               catInfoType := 'SURF';       {catalog type}
               catInfoCreator := 'TONY';    {catalog creator}
            END;
```

```
            WITH group1SrcTypes[2] DO
            BEGIN
               format := 'SPNT';            {SurfPaint document format}
               hint := 0;                   {no hint}
               flags := 0;                  {no flags}
               catInfoType := 'SPNT';       {catalog type}
               catInfoCreator := 'TONY';    {catalog creator}
            END;
            group1DstCount := 1;            {destination side has one type}
            group1DstEntrySize := SizeOf(FileTypeSpec);
            WITH group1DstTypes[1] DO
            BEGIN
               format := 'ttro';            {TeachText document format}
               hint := 0;                   {no hint}
               flags := taDstDocNeedsResourceFork;
                                            {TeachText documents need a }
                                            { resource fork (for pictures)}
               catInfoType := 'ttro';       {catalog type}
               catInfoCreator := 'ttxt';    {catalog creator}
            END;
         END; {WITH myPtr^}
      END; {IF}
   END; {IF}
   DoGetFileTranslationList := myErr;
END;
```

Because the list of file types that this extension can translate never changes, `DoGetFileTranslationList` fills out a file translation list the first time Macintosh Easy Open calls it; every other time it is called, `DoGetFileTranslationList` simply passes back the list it was passed.

In all likelihood, your translation extension will rely on external translators to perform the actual translation of files or scraps. If so, it's also likely that the user will be able to add and remove translators used by your extension—possibly by moving translators into or out of some specific folder. In that case, your `DoGetFileTranslationList` function could read the modification date of that folder and compare with a value you previously put in the `modDate` field to determine whether to regenerate the translation list.

Identifying Files

Once Macintosh Easy Open knows the types of files from and to which your extension can translate, it might call your extension to determine whether your extension can translate a particular file. This further check is necessary because some documents might have file types that are not specific enough for translation purposes. For example, a

document imported from a different operating system might have a file type of `'TEXT'`. Your translation extension might be able to determine, however, that the file actually contains SurfWriterPC data and hence deserves special format conversion treatment.

When your translation extension is called with the `kTranslateIdentifyFile` request code, your extension should identify the particular document. The `TranslateEntry` extension (shown in Listing 7-5 on page 7-25) dispatches to its `DoIdentifyFile` function when it receives this request code. Listing 7-7 shows the skeleton of a `DoIdentifyFile` function.

Listing 7-7 Identifying file types

```
FUNCTION DoIdentifyFile (self: ComponentInstance; theDoc: FSSpec;
                         VAR docKind: FileType): ComponentResult;
VAR
   isKnown: Boolean; {indicates whether this extension can identify the file}
BEGIN
   {call an extension-defined routine to do the real work}
   isKnown := MyIdentifyDocument(theDoc, docKind);
   IF isKnown THEN
      DoIdentifyFile := noErr
   ELSE
      DoIdentifyFile := noTypeErr;
END;
```

Some documents can be identified simply by inspecting their file type and creator. Other documents (in particular, those of type `'TEXT'`) might require opening the files and examining their contents to determine whether they can be translated by your extension. If your extension cannot recognize the document type, `DoIdentifyFile` should return `noTypeErr`. Otherwise, `DoIdentifyFile` should return `noErr`, and the `docKind` parameter should be set to the recognized file type.

Note
Your `DoIdentifyFile` function should not return `'TEXT'` as a file type unless it's certain that the document consists of plain, unformatted ASCII text. ◆

You should be aware that even if your extension identifies a particular document as one that it can translate, Macintosh Easy Open might not in fact call your extension to do the translation.

Translating Files

If your translation extension identifies a document as one that it can translate and the user chooses to use your translation extension, your extension is called with the `kTranslateTranslateFile` request code to translate the document. The

`TranslateEntry` extension (shown in Listing 7-5 on page 7-25) dispatches to its `DoTranslateFile` function when it receives this request code. Listing 7-8 shows the skeleton of a `DoTranslateFile` function.

Listing 7-8 Translating a document

```
FUNCTION DoTranslateFile (self: ComponentInstance;
                          refNum: TranslationRefNum;
                          srcDoc: FSSpec;
                          srcType: FileType;
                          srcTypeHint: LongInt;
                          dstDoc: FSSpec;
                          dstType: FileType;
                          dstTypeHint: LongInt): ComponentResult;
VAR
      myAdvert:      Handle;
      myResFile:     Integer;
      myResult:      OSErr;
CONST
   rProgressAdvertismentResID       = 150;
BEGIN
   myResFile := OpenComponentResFile(Component(self));
   IF myResFile <> -1 THEN
   BEGIN
      {get advertisement}
      myAdvert := Get1Resource('PICT', rProgressAdvertismentResID);
      DetachResource(myAdvert);
      {display progress dialog box and show advertisement}
      myResult := SetTranslationAdvertisement(refNum, PicHandle(myAdvert));
      myResult := CloseComponentResFile(myResFile);
   END;
   {now call your routine to translate the file}
   DoTranslateFile := MyDoTranslation
                                 (refNum, srcDoc, srcType, dstDoc, dstType);
   DisposeHandle(myAdvert);
END;
```

By the time the `DoTranslateFile` routine is called, the file specified by the `dstDoc` parameter already exists. The destination file has a data fork; it also has a resource fork if the `flags` field in the appropriate destination file type specification (in your extension's file translation list) has the `taDstDocNeedsResourceFork` bit set. Your extension should open the destination file and fill it with the translated data.

In Listing 7-8, the `DoTranslateFile` function calls the `SetTranslationAdvertisement` function to install an advertisement in the progress dialog box. The routine that does the actual data translation (`MyDoTranslation`) should periodically call `UpdateTranslationProgress` to update the progress bar in the dialog box.

If an error occurs during the translation, you should make sure to close any files you might have opened (for instance, the destination file's data fork and resource fork), do any other necessary cleaning up, and then return a nonzero result code through your component selector dispatcher. When Macintosh Easy Open receives a nonzero result code, it automatically deletes the destination file.

Writing Application Translation Extensions

Most applications can open only a certain number of file types and can therefore declare those openable file types by including an open resource in their resource forks. (See "Declaring the File Types Your Application Can Open" on page 7-13 for details about the open resource.) Some applications, however, need to determine dynamically which files they can open (perhaps because those applications already contain data-conversion capabilities using external filters). For these applications, the open resource alone is inadequate to specify which kinds of files they can open.

A simple way to generate dynamically a list of your application's openable file types is to provide an **application translation extension,** a translation extension that can create a list of file types and identify files, but which performs no actual translation. Essentially, the application translation extension exists solely to generate the dynamic list of file types your application can open. The source list in the file translation list that your extension returns to Macintosh Easy Open should contain a file type specification for each of those types; for the destination list of types, the file translation list should contain a single file type specification whose `format` field contains some arbitrary and otherwise unused file type. Suppose this destination file type is `'VOID'`.

The open resource in your application should then consist of a static list containing at least the value in the `format` field of the sole destination file type specification in the file translation list (that is, `'VOID'`). The net effect, as far as Macintosh Easy Open is concerned, is that your application can open documents of type `'VOID'` and that a translation extension exists that can translate some other file types into type `'VOID'`. As a result, the types in that list—which was generated dynamically—are now considered openable by your application.

Of course, in the situation imagined here, you don't want the application translation extension to do any actual data conversion. You indicate this by setting the `taDstIsAppTranslation` bit in the `flags` field of the destination file type specification. If this bit is set, Macintosh Easy Open gives the source document directly to your application without translation. No destination document is created.

Note
In the translation choices dialog box (illustrated in Figure 7-3 on page 7-6), a file type whose file type specification has the `taDstIsAppTranslation` bit set is listed by the application name only; the name of the application translation extension is not listed. ◆

Translation Manager Reference

This section describes the routines and resources that are specific to the Translation Manager. To take full advantage of the implicit translation capabilities of Macintosh Easy Open, you need to include appropriate resources in your application's resource fork. (See "Resources" beginning on page 7-43 for information on the open and kind resources.) In addition, you might need to call Translation Manager routines if your application doesn't use the Standard File Package or if it needs information about an application's translation capabilities.

IMPORTANT
The routines described in this section are intended for use by applications that bypass the Standard File Package or that need information about some application's ability to translate documents. Most applications don't need to use these routines. ▲

See "Translation Extension Reference" beginning on page 7-46 for information about data structures and routines you can use to write a translation extension.

IMPORTANT
The Translation Manager is not available in all operating environments. You should call the `Gestalt` function to ensure that it is available before calling any of its routines. See "Checking for the Translation Manager" on page 7-12 for details on calling `Gestalt`. ▲

Translation Manager Routines

The Translation Manager provides a number of routines that your application can call to get information about the documents and document types an application can open and to translate files and scraps. Normally, you need to use these routines only if your application doesn't use the Standard File Package to ask the user for names and locations of files to open, or if your application has other special needs.

Getting Translation Information

The Translation Manager provides several routines that you can use to get or set information about the file types that an application can open.

GetFileTypesThatAppCanNativelyOpen

You can use the `GetFileTypesThatAppCanNativelyOpen` function to obtain a list of file types that an application can open by itself.

```
FUNCTION GetFileTypesThatAppCanNativelyOpen
                  (appVRefNumHint: Integer; appSignature: OSType;
                   VAR nativeTypes: TypesBlock): OSErr;
```

`appVRefNumHint`
The volume reference number of volume containing the application. The search for the specified application begins on this volume; if the application isn't found there, the search continues to other mounted volumes.

`appSignature`
The signature of the application.

`nativeTypes`
On exit, a zero-terminated file types that the application can open without translation.

DESCRIPTION

The `GetFileTypesThatAppCanNativelyOpen` function returns, through the `nativeTypes` parameter, a list of all the file types that can be opened by the application having the signature `appSignature`. If `GetFileTypesThatAppCanNativelyOpen` returns successfully, the `nativeTypes` parameter contains a list of up to 64 file types. The structure of the list is defined by the `TypesBlock` data type.

```
TYPE
   TypesBlock     = ARRAY[0..63] OF FileType;
   TypesBlockPtr  = ^TypesBlock;
```

If fewer than 64 types are returned, the end of the list is indicated by an entry whose value is 0.

SPECIAL CONSIDERATIONS

The `GetFileTypesThatAppCanNativelyOpen` function is not available in all versions of system software; use the `Gestalt` function to determine whether the Translation Manager is available before calling it.

The `GetFileTypesThatAppCanNativelyOpen` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `GetFileTypesThatAppCanNativelyOpen` are

Trap macro	Selector
`_TranslationDispatch`	$001C

RESULT CODES

`noErr`	0	No error
`wrgVolTypErr`	–123	Volume does not support Desktop Manager
`afpItemNotFound`	–5012	Information not found

ExtendFileTypeList

You can use the `ExtendFileTypeList` function to create a list of file types that can be translated into a type in a given list. The Standard File Package calls this function internally; your application probably won't need to use it.

```
FUNCTION ExtendFileTypeList (originalTypeList: FileTypePtr;
                             numberOriginalTypes: Integer;
                             extendedTypeList: FileTypePtr;
                             VAR numberExtendedTypes: Integer)
                             : OSErr;
```

`originalTypeList`
A pointer to a list of file types.

`numberOriginalTypes`
The number of file types in `originalTypeList`.

`extendedTypeList`
On exit, a pointer to a list of file types that can be translated into the types in `originalTypeList`.

`numberExtendedTypes`
On entry, the maximum number of file types that can be put into the `extendedTypeList` parameter. On exit, the actual number of file types put into the extended type list.

DESCRIPTION

The `ExtendFileTypeList` function takes the set of types in the `originalTypeList` parameter and returns (in the `extendedTypeList` parameter) a list of types that can be translated into those types. The `extendedTypeList` parameter is of type `FileTypePtr`, which is a pointer to a file type.

```
TYPE
   FileTypePtr    = ^FileType;
```

Note that the number of types specified in the parameters `numberOriginalTypes` and `numberExtendedTypes` is limited only by available memory.

SPECIAL CONSIDERATIONS

The `ExtendFileTypeList` function is not available in all versions of system software; use the `Gestalt` function to determine whether the Translation Manager is available before calling it.

The `ExtendFileTypeList` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ExtendFileTypeList` procedure are

Trap macro	Selector
`_TranslationDispatch`	$0009

RESULT CODE

`noErr`	0	No error

CanDocBeOpened

You can use the `CanDocBeOpened` function to determine whether a specified application can open a particular document.

```
FUNCTION CanDocBeOpened
                  (targetDocument: FSSpec;
                   appVRefNumHint: Integer;
                   appSignature: OSType;
                   nativeTypes: TypesBlockPtr;
                   onlyNative: Boolean;
                   VAR howToOpen: DocOpenMethod;
                   VAR howToTranslate: FileTranslationSpec)
                   : OSErr;
```

`targetDocument`
The document to check.

`appVRefNumHint`
The volume reference number of the volume containing the application. The search for the specified application begins on this volume; if the application isn't found there, the search continues to other mounted volumes.

`appSignature`
The signature of the application.

`nativeTypes`
A zero-terminated list of file types that the application can open without translation; if this parameter contains `NIL`, the default list of file types returned by the `GetFileTypesThatAppCanNativelyOpen` function is used.

`onlyNative`
If `TRUE`, determine only whether the application can open the document without translation; otherwise, determine whether the application can open the document after translation.

`howToOpen` On exit, the method of opening the document. This field contains a meaningful value only if `CanDocBeOpened` returns `noErr` (indicating that the specified document can be opened).

`howToTranslate`
On exit, a buffer of information (in a private format) indicating how to translate the document.

DESCRIPTION

The `CanDocBeOpened` function determines whether a document can be opened by a particular application. If the application can open the document, `CanDocBeOpened` returns the result code `noErr` and sets the `howToOpen` parameter to a constant that indicates the method that the application would use to open the document. The `howToOpen` parameter contains a document-opening method:

```
TYPE DocOpenMethod =
      (domCannot, domNative, domTranslateFirst, domWildcard);
```

The `domCannot` constant indicates that the application cannot open the document. The `domNative` constant indicates that the application can open the document natively. The `domTranslateFirst` constant indicates that the application can open the document only after it's been translated. The `domWildcard` constant indicates that the application has the file type `'****'` in its list of the file types that it can open and hence can open any type of document.

If the document needs to be translated before it can be opened (as indicated by the `domTranslateFirst` method), `CanDocBeOpened` returns in the `howToTranslate` parameter a buffer of information indicating how to translate the document. The format of this information is private.

```
TYPE
   FileTranslationSpec     = ARRAY[1..12] OF LongInt;
```

You pass the information returned in the `howToTranslate` parameter to the `TranslateFile` function.

SPECIAL CONSIDERATIONS

A preference must have already been set (using the Document Converter tool) on how to open the document.

The `CanDocBeOpened` function is not available in all versions of system software; use the `Gestalt` function to determine whether the Translation Manager is available before calling it.

The `CanDocBeOpened` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `CanDocBeOpened` procedure are

Trap macro	Selector
`_TranslationDispatch`	$001E

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	File not found
`paramErr`	–50	Parameter error
`extFSErr`	–58	External file system
`dirNFErr`	–120	Directory not found or incomplete pathname
`badTranslationSpecErr`	–3031	Translation path is invalid
`noPrefAppErr`	–3032	No translation preference available
`afpItemNotFound`	–5012	Information not found

Translating Files

The Translation Manager provides a routine that you can use to translate files.

TranslateFile

You can use `TranslateFile` to translate a document from one format to another.

```
FUNCTION TranslateFile (sourceDocument: FSSpec;
                        destinationDocument: FSSpec;
                        howToTranslate: FileTranslationSpec)
                        : OSErr;
```

`sourceDocument`
The document to translate.

`destinationDocument`
The file to put the translated document into.

`howToTranslate`
A buffer of information indicating how to translate the document.

DESCRIPTION

The `TranslateFile` function reads the file specified by the `sourceDocument` parameter and translates it into another format. You specify in the `destinationDocument` parameter the name and location of a file to contain the translated data. Note that your application only specifies the name and location for the file; `TranslateFile` creates the file and puts the translated data into it. The destination file must not exist before you call `TranslateFile`.

The translation is performed according to the information provided in the `howToTranslate` parameter. Usually, you'll get that information by calling `CanDocBeOpened`.

SPECIAL CONSIDERATIONS

The `TranslateFile` function is not available in all versions of system software; use the `Gestalt` function to determine whether the Translation Manager is available before calling it.

The `TranslateFile` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `TranslateFile` procedure are

Trap macro	Selector
`_TranslationDispatch`	$000C

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	Directory full
`dskFulErr`	–34	Disk full
`nvsErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`tmfoErr`	–42	Too many files open
`fnfErr`	–43	File not found
`wPrErr`	–44	Disk is write protected
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Volume is locked
`dupFNErr`	–48	Duplicate filename (rename)
`opWrErr`	–49	File already open with write permission
`extFSErr`	–58	External file system
`dirNFErr`	–120	Directory not found
`userCanceledErr`	–128	User canceled
`badTranslationSpecErr`	–3031	`howToTranslate` is invalid

Resources

This section describes the resources used by the Translation Manager.

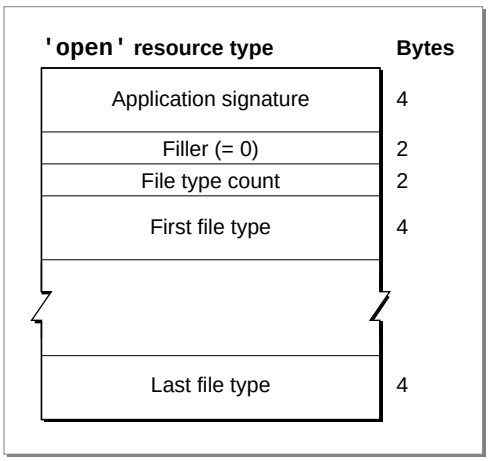
- The `'open'` resource indicates which kinds of documents your application can open.
- The `'kind'` resource defines a custom kind string for your application's documents.

Information from these resources is stored in a volume's desktop database. Any changes made to an application's open or kind resources won't appear until the desktop is rebuilt.

The Open Resource

To make your application compatible with the Translation Manager, you should add an open resource to your application's resource file. This resource, of type `'open'`, indicates which types of files your application can open. Figure 7-11 shows the format of a compiled open resource.

Figure 7-11 Structure of a compiled open (`'open'`) resource



An open resource consists of your application's signature and a list of file types. The Finder allows the user to launch your application by dropping documents of any of those types on your application's icon. In addition, if any translation extensions are installed, all documents that can be translated into one of those file types can also be dropped onto your application's icon. Your application's open resource having resource ID 128 is used by the Standard File Package routine `StandardOpenDialog` to determine the file types displayed in the standard file-opening dialog box.

IMPORTANT

The file types in the open resource should be ordered by preference. If the Translation Manager has to choose between multiple file types as the destination file type for a translation, it chooses the file type that occurs earliest in the list. ▲

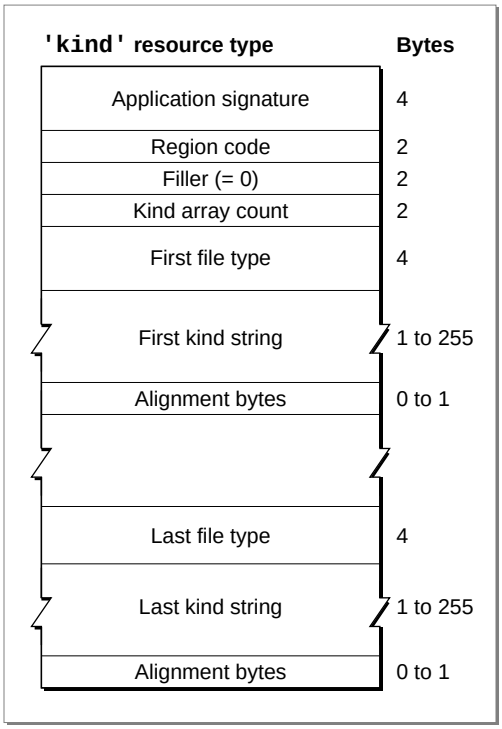
Because your application's signature is included in the open resource, the resource can be in some file other than the application's resource fork. However, an open resource located in an application's resource fork overrides any open resource for that application located elsewhere. It also overrides the openable file types as listed in the application's `'FREF'` resource.

See Listing 7-2 on page 7-13 for a sample open resource.

The Kind Resource

You should add a kind resource to your application's resource file. This resource, of type `'kind'`, specifies custom kind strings, which override the Finder's normal algorithm for generating kind strings. Figure 7-12 shows the format of a compiled kind resource.

Figure 7-12 Structure of a compiled kind (`'kind'`) resource



A kind resource consists of your application's signature and a list of file types and their associated custom kind strings. The Finder displays a document's kind string when a folder's contents are viewed by name, size, kind, label, or date (that is, by any method other than by icon or small icon).

Because your application's signature is included in the kind resource, the resource can be located in some file other than the application's resource fork. However, a kind resource located in an application's resource fork overrides any kind resource for that application located elsewhere.

A kind resource contains a region code, which specifies the region code of the kind strings contained in the resource. The Finder uses only custom kind strings that have the same region code as the current system itself.

In the list of file types and associated kind strings, you can use the special file type `ftApplicationName` to specify the name of your application. Whenever Macintosh Easy Open encounters a document that belongs to your application but whose file type

isn't listed in your application's kind resource, the Finder uses its standard algorithm to generate a kind string in the form "<application name> document".

Note
See Listing 7-3 on page 7-15 for a sample kind resource. ◆

Translation Extension Reference

This section describes the data structures and routines you can use to write a translation extension. It also describes the routines that your translation extension has to contain.

See "Translation Manager Reference" beginning on page 7-36 for a description of the routines and resources that are specific to the Translation Manager.

Translation Extension Data Structures

This section describes the data structures you'll need to use when writing a translation extension.

File Type Specifications

You use file type specifications to describe document formats in a file translation list. (See the next section for a description of file translation lists.) The interpretation of some of the fields of a file type specification depends on whether the specification occurs in the list of source document types or in the list of destination document types. A file type specification is defined by the `FileTypeSpec` data structure.

```
TYPE FileTypeSpec =
   RECORD
      format:           FileType;
      hint:             LongInt;
      flags:            TranslationAttributes;
      catInfoType:      OSType;
      catInfoCreator:   OSType;
   END;
```

Field descriptions

`format` The translation file type of the document. Macintosh Easy Open uses this field as the canonical way to describe the format of a file for translation purposes.

`hint` A 4-byte value reserved for use by your translation extension.

`flags` A 4-byte value consisting of bit flags that specify how to control the translation. This field is used only for destination file types; you should set it to 0 for all source file type specifications. Currently 2 bits are defined; all other bits should be cleared to 0:

```
CONST
   taDstDocNeedsResourceFork     = 1;
   taDstIsAppTranslation         = 2;
```

Before Macintosh Easy Open sends your translation extension the `kTranslateTranslateFile` request code, it has already created the destination file's data fork. The bit specified by the constant `taDstDocNeedsResourceFork` should be set if the translated document also needs a resource fork.

The bit specified by the constant `taDstIsAppTranslation` should be set if your extension doesn't actually perform the translation because an associated application can already translate the specified file type into the desired format. See "Writing Application Translation Extensions" on page 7-35 for more details.

`catInfoType` The type of the file as contained in the volume's catalog file.

`catInfoCreator`

The creator of the file as contained in the volume's catalog file.

In file type specifications occurring in the list of source document types in a file translation list, Macintosh Easy Open uses the `format` and `catInfoCreator` fields to determine the kind string displayed in the "From" format specification of the translation progress dialog box (see Figure 7-4 on page 7-7).

In file type specifications occurring in the list of destination document types in a file translation list, Macintosh Easy Open uses the `format` and `catInfoCreator` fields to determine the kind string displayed in the "To" format specification in the translation progress dialog box (see Figure 7-4 on page 7-7). The `format` and `catInfoCreator` fields are also used to get the information displayed in the Document Converter dialog box (Figure 7-7 on page 7-9). However, Macintosh Easy Open uses the `catInfoType` and `catInfoCreator` fields to set the catalog type and creator of the destination file.

Note

See page 7-19 for a discussion of why the translation file type described in the `format` field may differ from the catalog type described in the `catInfoType` field. ◆

File Translation Lists

You use the `FileTranslationList` data structure to describe which file formats your extension can translate into which other file formats. The Translation Manager uses the file translation list that it gets from each translation system to create a master database of format translations it can direct.

```
TYPE FileTranslationList =
   RECORD
      modDate:                LongInt;
      groupCount:             LongInt;
      {group1SrcCount:        LongInt;}
      {group1SrcEntrySize:    LongInt;}
      {group1SrcTypes:        ARRAY[1..group1SrcCount] OF FileTypeSpec;}
      {group1DstCount:        LongInt;}
      {group1DstEntrySize:    LongInt;}
      {group1DstTypes:        ARRAY[1..group1DstCount] OF FileTypeSpec;}
      {repeat above six lines for a total of groupCount times}
   END;
   FileTranslationListPtr     = ^FileTranslationList;
   FileTranslationListHandle  = ^FileTranslationListPtr;
```

A file translation list consists of a field indicating the modification date of the list and a count of the number of groups that follow those two fields. The size of the translation list prepared by an extension is variable, depending upon the number of groups, the file specification record size, and the number of file types that the extension knows about.

Field descriptions

`modDate` The creation date of the file translation list. If your extension uses external translators, you might set this field to the modification date of a folder containing those translators.

`groupCount` The number of translation groups that follow.

`group1SrcCount` The number of file types that the extension can read in a group.

`group1SrcEntrySize` The size of the file specification records in the array that follows this field. In general, you can set this field to `SizeOf(FileTypeSpec)`.

`group1SrcTypes` An array of file specification records. You should include a file specification record in this array for each file type that your extension knows how to translate.

`group1DstCount` The number of file types that the extension can write in a group.

`group1DstEntrySize`
The size of the file specification records in the array that follows this field. In general, you can set this field to `SizeOf(FileTypeSpec)`.

`group1DstTypes`
An array of file specification records. You should include a file specification record in this array for each file type that your extension can translate into.

Scrap Type Specifications

You use the `ScrapTypeSpec` data structure to describe a specific scrap format.

```
TYPE ScrapTypeSpec =
   RECORD
      format:           ScrapType;
      hint:             LongInt;
   END;
```

Field descriptions

`format` The type of the specified scrap.

`hint` A 4-byte value reserved for use by your translation extension.

Scrap Translation Lists

You use the `ScrapTranslationList` data structure to describe which scrap formats your extension can translate into which other scrap formats. The Translation Manager uses the **scrap translation list** that it gets from each translation system to create a master database of its translation capability.

```
TYPE ScrapTranslationList =
   RECORD
      modDate:                LongInt;
      groupCount:             LongInt;
      {group1SrcCount:        LongInt;}
      {group1SrcEntrySize:    LongInt;}
      {group1SrcTypes:        ARRAY[1..group1SrcCount] OF ScrapTypeSpec;}
      {group1DstCount:        LongInt;}
      {group1DstEntrySize:    LongInt;}
      {group1DstTypes:        ARRAY[1..group1DstCount] OF ScrapTypeSpec;}
      {repeat above six lines for a total of groupCount times}
   END;
   ScrapTranslationListPtr    = ^ScrapTranslationList;
   ScrapTranslationListHandle = ^ScrapTranslationListPtr;
```

A scrap translation list consists of a field indicating the modification date of the list and a count of the number of groups that follow those two fields. The size of the translation list prepared by an extension is variable, depending upon the number of groups, the scrap specification record size, and the number of scrap types that the extension knows about.

Field descriptions

`modDate` The creation date of the scrap translation list. If your extension uses external translators, you might set this field to the modification date of a folder containing those translators.

`groupCount` The number of translation groups that follow.

`group1SrcCount` The number of scrap types that the extension can read in a group.

`group1SrcEntrySize` The size of the scrap specification records in the array that follows this field. In general, you can set this field to `SizeOf(ScrapTypeSpec)`.

`group1SrcTypes` An array of scrap specification records. You should include a scrap specification record in this array for each scrap type that your extension knows how to translate.

`group1DstCount` The number of scrap types that the extension can write in a group.

`group1DstEntrySize` The size of the scrap specification records in the array that follows this field. In general, you can set this field to `SizeOf(ScrapTypeSpec)`.

`group1DstTypes` An array of scrap specification records. You should include a scrap specification record in this array for each scrap type that your extension can translate into.

Translation Extension Routines

This section describes two routines that you can call from within a translation extension.

Managing Translation Progress Dialog Boxes

You can use the `SetTranslationAdvertisement` function to display the progress dialog box and, optionally, to include a logo or other identifying picture in the progress dialog box. You can use the `UpdateTranslationProgress` function to show the user the progress of a translation and allow the user to cancel a translation.

SetTranslationAdvertisement

A translation extension can call `SetTranslationAdvertisement` to install an advertisement into the progress dialog box.

```
FUNCTION SetTranslationAdvertisement (refNum: TranslationRefNum;
                                      advertisement: PicHandle)
                                      : OSErr;
```

`refNum` A translation reference number.

`advertisement`
A handle to a picture to display in the upper portion of the dialog box.

DESCRIPTION

The `SetTranslationAdvertisement` function installs a translation extension-specific picture into the upper portion of a translation progress dialog box, then displays the dialog box. The `advertisement` parameter should be a handle to the picture to display. If the value of `advertisement` is `NIL`, no advertisement is displayed and the upper portion of the dialog box is removed before the box is displayed to the user.

Your translation extension can read the picture data from its resource fork, but it should detach the resource from the resource fork (by calling `DetachResource`) and make the handle unpurgeable before calling `SetTranslationAdvertisement`. Because you'll usually load the picture data into the temporary heap provided for the translation extension, the picture data is automatically disposed of when that heap is destroyed. If your translation extension loads the picture data elsewhere in memory, you are responsible for disposing of it before returning from your `DoTranslateFile` or `DoTranslateScrap` routine.

The size of the picture to display can be no larger than 280 by 50 pixels. If the picture you specify is smaller than that, it is automatically centered (both vertically and horizontally) in the available space.

You should set the `refNum` parameter to the translation reference number passed to your `DoTranslateFile` or `DoTranslateScrap` routine. The Translation Manager uses that number internally.

SPECIAL CONSIDERATIONS

Your translation extension should call `SetTranslationAdvertisement` only in response to the `kTranslateTranslateFile` or `kTranslateTranslateScrap` request code (that is, in your `DoTranslateFile` or `DoTranslateScrap` routine). Do not call this function in response to any other request code or from any code that isn't a translation extension.

You must call `SetTranslationAdvertisement` before you call the `UpdateTranslationProgress` procedure for the first time.

The `SetTranslationAdvertisement` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `SetTranslationAdvertisement` function are

Trap macro	Selector
`_TranslationDispatch`	$0002

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter error
`memFullErr`	–108	Not enough memory

SEE ALSO

See Figure 7-4 on page 7-7 for a sample translation progress dialog box showing an advertisement. See Listing 7-8 on page 7-34 for an example of the use of `SetTranslationAdvertisement`.

UpdateTranslationProgress

A translation extension can call `UpdateTranslationProgress` to update the progress dialog box that is displayed during file and scrap translation and to give the user a chance to click the Cancel button.

```
FUNCTION UpdateTranslationProgress (refNum: TranslationRefNum;
                                    percentDone: Integer;
                                    VAR canceled: Boolean)
                                    : OSErr;
```

`refNum` A translation reference number.

`percentDone`
An integer in the range 0–100 that indicates the percentage of the translation that has been completed.

`canceled` On exit, `UpdateTranslationProgress` returns `TRUE` in this parameter if the user clicked the Cancel button in the progress dialog box; otherwise, `UpdateTranslationProgress` returns `FALSE` in this parameter.

DESCRIPTION

The `UpdateTranslationProgress` function updates the translation progress dialog box. You should call this function periodically during a translation to update the progress bar and to give the user an opportunity to cancel the translation. If the user clicks the Cancel button in the dialog box (or types Command-period while the box is displayed), the `canceled` parameter is set to `TRUE`; otherwise, it is set to `FALSE`. When `canceled` returns `TRUE`, you should stop the translation, and your application-defined routine `DoTranslateFile` or `DoTranslateScrap` should return the result code `userCancelledErr`.

The `percentDone` parameter specifies the approximate percentage of time elapsed until completion. You should call `UpdateTranslationProgress` periodically at reasonable time intervals to allow the user to cancel the translation. When the translation is complete, you should call `UpdateTranslationProgress` with `percentDone` set to 100 so that the user can see that the translation is complete.

You should set the `refNum` parameter to the translation reference number passed to your `DoTranslateFile` or `DoTranslateScrap` routine. The Translation Manager uses that number internally.

SPECIAL CONSIDERATIONS

Your translation extension should call `UpdateTranslationProgress` only in response to the `kTranslateTranslateFile` or `kTranslateTranslateScrap` request code (that is, in your `DoTranslateFile` or `DoTranslateScrap` routine). Do not call this function in response to any other request code or from any code that isn't a translation extension.

You should already have called `SetTranslationAdvertisement` before calling `UpdateTranslationProgress`.

The `UpdateTranslationProgress` function might cause memory to be moved or purged; you should not call it at interrupt time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `UpdateTranslationProgress` function are

Trap macro	Selector
`_TranslationDispatch`	$0001

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter error
`memFullErr`	–108	Not enough memory

Translation Extension-Defined Routines

This section describes the routines you'll need to define in order to write a translation extension. You can create both file and scrap translation extensions.

To construct a translation extension to translate files, you need to create a component that responds to the `kTranslateGetFileTranslationList`, `kTranslateIdentifyFile`, and `kTranslateTranslateFile` request codes. In response to these request codes, you typically dispatch to one of the extension-defined routines `DoGetFileTranslationList`, `DoIdentifyFile`, and `DoTranslateFile`. To construct a translation extension that translates scraps, you need to create a component that responds to the `kTranslateGetScrapTranslationList`, `kTranslateIdentifyScrap`, and `kTranslateTranslateScrap` request codes. In response to these request codes, you typically dispatch to one of the extension-defined routines `DoGetScrapTranslationList`, `DoIdentifyScrap`, and `DoTranslateScrap`.

All routines return result codes. If they succeed, they should return `noErr`. The Component Manager requires these routines to return a value of type `ComponentResult`—a value of type `LongInt`—to simplify dispatching.

See "Dispatching to Translation Extension-Defined Routines" beginning on page 7-24 for a description of how you call these routines from within a translation extension.

File Translation Extension Routines

To write a file translation extension, you need to define three routines:

- `DoGetFileTranslationList`
- `DoIdentifyFile`
- `DoTranslateFile`

DoGetFileTranslationList

A file translation extension must respond to the `kTranslateGetFileTranslationList` request code. Whenever it first notices the extension, Macintosh Easy Open calls your extension with this request code to obtain a

list of the file types that the extension can translate. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoGetFileTranslationList` function.

```
FUNCTION DoGetFileTranslationList
                  (self: ComponentInstance;
                   translationList: FileTranslationListHandle)
                   : ComponentResult;
```

`self` A component instance that identifies the component containing the translation extension.

`translationList`
A handle to a file translation list.

DESCRIPTION

Your `DoGetFileTranslationList` function should return, through the `translationList` parameter, a handle to a list of the file types from and into which your translation extension can translate. On entry to `DoGetFileTranslationList`, the `translationList` parameter contains a handle to a structure of type `FileTranslationList`. If your translation extension can translate any files at all, your `DoGetFileTranslationList` function should resize that handle and fill the block with a list of the file types it can translate. If the translation list whose handle you return in `translationList` has the `groupCount` field set to 0, Macintosh Easy Open assumes that your extension cannot translate any file types.

Macintosh Easy Open calls your translation extension's `DoGetFileTranslationList` function when it first becomes aware of your extension. For improved performance, Macintosh Easy Open remembers each translation extension's most recently returned file translation list and passes that list to `DoGetFileTranslationList` in the `translationList` parameter. If you determine that the list hasn't changed, you should simply return the same handle to Macintosh Easy Open.

RESULT CODES

The `DoGetFileTranslationList` function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

See "File Translation Lists" on page 7-48 for a description of the `FileTranslationList` data structure. See "Writing a Translation Extension" beginning on page 7-18 for more information about implementing a translation extension. See Listing 7-6 on page 7-30 for a routine that constructs a sample file translation list.

DoIdentifyFile

A file translation extension must respond to the `kTranslateIdentifyFile` request code. The Translation Manager uses this request code to allow the translation extension to identify a file as having a format that the extension can translate. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoIdentifyFile` function.

```
FUNCTION DoIdentifyFile (self: ComponentInstance;
                         theDoc: FSSpec;
                         VAR docKind: FileType)
                         : ComponentResult;
```

`self` A component instance that identifies the component containing the translation extension.

`theDoc` A file system specification record that specifies the document that the translation extension must identify.

`docKind` On exit, the file format type of the document as identified by your translation extension.

DESCRIPTION

Your `DoIdentifyFile` function returns, through the `docKind` parameter, the file type of the file specified by the `FSSpec` record passed in the `theDoc` parameter. If your translation extension does not recognize the type of the specified file, `DoIdentifyFile` should return the result code `noTypeErr`.

SPECIAL CONSIDERATIONS

Your `DoIdentifyFile` function should not return `'TEXT'` as a file type unless you determine that the document consists solely of a plain, unformatted stream of ASCII characters.

RESULT CODES

`noErr`	0	No error
`noTypeErr`	–102	Unrecognized file type

DoTranslateFile

A file translation extension must respond to the `kTranslateTranslateFile` request code. The Translation Manager uses this request code to allow the translation extension to translate a file from one format to another. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoTranslateFile` function.

```
FUNCTION DoTranslateFile (self: ComponentInstance;
                          refNum: TranslationRefNum;
                          srcDoc: FSSpec;
                          srcType: FileType;
                          srcTypeHint: LongInt;
                          dstDoc: FSSpec;
                          dstType: FileType;
                          dstTypeHint: LongInt): ComponentResult;
```

`self` A component instance that identifies the component containing your translation extension.

`refNum` The translation reference number for this translation.

`srcDoc` A file system specification record that specifies the source document.

`srcType` The format of the file to be translated.

`srcTypeHint` The value in the `hint` field of the source document's file type specification.

`dstDoc` A file system specification record that specifies the destination document.

`dstType` The format into which to translate the source document.

`dstTypeHint` The value in the `hint` field of the destination document's file type specification.

DESCRIPTION

Your `DoTranslateFile` function translates a document from one format into another. The document to be translated is specified by the `srcDoc` parameter, and your routine should put the translated document into the file specified by the `dstDoc` parameter. The data fork of the destination file already exists by the time `DoTranslateFile` is called. In addition, if the `flags` field in the appropriate destination file type specification in your extension's file translation list has the `taDstDocNeedsResourceFork` bit set, the destination file already contains a resource fork. Your function should open the destination file and fill its data or resource fork (or both) with the appropriate translated data.

The `refNum` parameter is a reference number that Macintosh Easy Open assigns to the translation. Each translation is assigned a unique number to distinguish the translation from any other translations that might occur. You need to pass this reference number to any Macintosh Easy Open routines you call from within the file translation extension; for instance, if by calling the `SetTranslationAdvertisement` function you display the progress dialog box, you'll pass that reference number in the `refNum` parameter.

The `DoTranslateFile` function can translate the source file itself or rely upon external translators. If it cannot translate the source file, your function should return a result code different from `noErr`. In that case, Macintosh Easy Open will automatically delete the destination file.

Your translation extension should call the `SetTranslationAdvertisement` function to display the progress dialog box and the `UpdateTranslationProgress` function to update the dialog box periodically.

Your `DoTranslateFile` function should return `noErr` if successful or an appropriate result code otherwise.

RESULT CODES

`noErr`	0	No error
`userCanceledErr`	–128	User canceled the translation
`invalidTranslationPathErr`	–3025	`srcType` to `dstType` is not a valid translation path
`couldNotParseSourceFileErr`	–3026	The source document is not of type `srcType`

Scrap Translation Extension Routines

To write a scrap translation extension, you need to define three routines:

- `DoGetScrapTranslationList`
- `DoIdentifyScrap`
- `DoTranslateScrap`

DoGetScrapTranslationList

A scrap translation extension must respond to the `kTranslateGetScrapTranslationList` request code. At system startup time, the Translation Manager calls your extension with this request code to obtain a list of the

scrap types that the extension can translate. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoGetScrapTranslationList` function.

```
FUNCTION DoGetScrapTranslationList
                        (self: ComponentInstance;
                         list: ScrapTranslationListHandle)
                         : ComponentResult;
```

`self` A component instance that identifies the component containing the translation extension.

`list` A handle to a scrap translation list.

DESCRIPTION

The `DoGetScrapTranslationList` function returns, through the `list` parameter, a handle to a list of the scrap types from and into which your translation extension can translate. On entry to `DoGetScrapTranslationList`, the `list` parameter contains a handle to a structure of type `ScrapTranslationList`. If your translation extension can translate any scrap types at all, your `DoGetScrapTranslationList` function should resize that handle and fill the block with a list of the scrap types it can translate. If the translation list whose handle you return in `list` has the `groupCount` field set to 0, Macintosh Easy Open assumes that your extension cannot translate any scrap types.

When it first becomes aware of your extension, Macintosh Easy Open calls your translation extension's `DoGetScrapTranslationList` function. For improved performance, Macintosh Easy Open remembers each translation extension's most recently returned scrap translation list and passes that list to `DoGetScrapTranslationList` in the `list` parameter. If you determine that the list hasn't changed, you should simply return the same handle to Macintosh Easy Open.

RESULT CODES

The `DoGetScrapTranslationList` function should return `noErr` if successful, or an appropriate result code otherwise.

SEE ALSO

See "Scrap Translation Lists" on page 7-49 for a description of the `ScrapTranslationList` data structure. See "Writing a Translation Extension" beginning on page 7-18 for more information about implementing a translation extension.

DoIdentifyScrap

A scrap translation extension must respond to the `kTranslateIdentifyScrap` request code. The Translation Manager uses this request code to allow the translation extension to identify a scrap as one that the extension can translate. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoIdentifyScrap` function.

```
FUNCTION DoIdentifyScrap (self: ComponentInstance;
                          dataPtr: Ptr;
                          dataLength: Size;
                          VAR dataFormat: ScrapType)
                          : ComponentResult;
```

`self` A component instance that identifies the component containing the translation extension.

`dataPtr` A pointer to the scrap.

`dataLength`
The size of the scrap to be translated.

`dataFormat`
On entry, the type of the scrap format. On exit, the type of the scrap format as recognized by your translation extension.

DESCRIPTION

Your `DoIdentifyScrap` function returns, through the `dataFormat` parameter, the scrap type of the scrap specified by the `dataPtr` and `dataLength` parameters. If your translation extension does not recognize the type of the specified scrap, `DoIdentifyScrap` should return the result code `noTypeErr`.

In general, the scrap that your `DoIdentifyScrap` function is asked to identify is always in one of the formats listed among the source formats in the translation groups contained in your extension's scrap translation list. Your scrap translation extension therefore needs only to verify that the indicated scrap is of the specified format.

RESULT CODES

`noErr`	0	No error
`noTypeErr`	–102	Unrecognized scrap type

DoTranslateScrap

A scrap translation extension must respond to the `kTranslateTranslateScrap` request code. The Translation Manager sends this request code to allow the extension to translate scraps from one format to another. You can handle this request by calling the `CallComponentFunctionWithStorage` function and passing it a pointer to a function with the syntax defined by the `DoTranslateScrap` function.

```
FUNCTION DoTranslateScrap (self: ComponentInstance;
                           refNum: TranslationRefNum;
                           srcDataPtr: Ptr;
                           srcDataLength: Size;
                           srcType: ScrapType;
                           srcTypeHint: LongInt;
                           dstData: Handle;
                           dstType: ScrapType;
                           dstTypeHint: LongInt)
                           : ComponentResult;
```

`self` A component instance that identifies the component containing the translation extension.

`refNum` The translation reference number for this translation.

`srcDataPtr`
A pointer to the scrap to be translated.

`srcDataLength`
The size of the scrap to be translated.

`srcType` The format of the scrap to be translated.

`srcTypeHint`
The value in the `hint` field of the source document's scrap type specification.

`dstData` A handle to the destination to be filled in.

`dstType` The format into which to translate the source scrap.

`dstTypeHint`
The value in the `hint` field of the destination document's scrap type specification.

DESCRIPTION

The `DoTranslateScrap` function translates a scrap from one format into another. The scrap to be translated is specified by the `srcDataPtr` and `srcDataLength` parameters, and your routine should put the translated data into the block specified by the `dstData` parameter. Your function should resize that block as necessary and fill it with the appropriate translated data.

The `refNum` parameter is a reference number that Macintosh Easy Open assigns to the translation. Each translation is assigned a unique number to distinguish the translation from any other translations that might be occurring. You need to pass this reference number to any Macintosh Easy Open routines you call from within the scrap translation extension; for instance, if you display the progress dialog box by calling the `SetTranslationAdvertisement` function, you'll pass that reference number in the `refNum` parameter.

The `DoTranslateScrap` function can translate the source file itself or rely upon external translators. If it cannot translate the source scrap, your function should return a result code different from `noErr`.

Your translation extension should call the `SetTranslationAdvertisement` function to display the progress dialog box and the `UpdateTranslationProgress` function to update the dialog box periodically.

RESULT CODES

The `DoTranslateScrap` function should return `noErr` if successful, or an appropriate result code otherwise.

Summary of the Translation Manager

This section provides Pascal, C, and assembly-language summaries for the constants, data types, and routines provided by the Translation Manager for use by applications. For a summary of the constants, data types, and routines that you can use or need to define if you're writing a translation extension, see "Summary of Translation Extensions" beginning on page 7-68.

Pascal Summary

Constants

```
CONST
   {Gestalt selectors and response bit numbers}
   gestaltTranslationAttr                = 'xlat';   {Translation Manager}
   gestaltTranslationMgrExists           = 0;        {TM is present}

   gestaltStandardFileAttr               = 'stdf';   {Standard File Package}
   gestaltStandardFile58                 = 0;
   gestaltStandardFileTranslationAware   = 1;
   gestaltStandardFileHasColorIcons      = 2;

   gestaltEditionMgrAttr                 = 'edtn';   {Edition Manager}
   gestaltEditionMgrPresent              = 0;
   gestaltEditionMgrTranslationAware     = 1;

   gestaltScrapMgrAttr                   = 'scra';   {Scrap Manager}
   gestaltScrapMgrTranslationAware       = 0;
```

Data Types

```
TYPE
   FileType               = OSType;                {file types}
   ScrapType              = ResType;               {scrap types}

   FileTypePtr            = ^FileType;

   FileTranslationSpec    = ARRAY[1..12] OF LongInt;
```

```
TypesBlock              = ARRAY[0..63] OF FileType;
TypesBlockPtr           = ^TypesBlock;

DocOpenMethod           = (domCannot,
                           domNative,
                           domTranslateFirst,
                           domWildcard);
```

Translation Manager Routines

Getting Translation Information

```
FUNCTION GetFileTypesThatAppCanNativelyOpen
                           (appVRefNumHint: Integer; appSignature: OSType;
                            VAR nativeTypes: TypesBlock): OSErr;
FUNCTION ExtendFileTypeList
                           (originalTypeList: FileTypePtr;
                            numberOriginalTypes: Integer;
                            extendedTypeList: FileTypePtr;
                            VAR numberExtendedTypes: Integer): OSErr;
FUNCTION CanDocBeOpened    (targetDocument: FSSpec;
                            appVRefNumHint: Integer;
                            appSignature: OSType;
                            nativeTypes: TypesBlockPtr;
                            onlyNative: Boolean;
                            VAR howToOpen: DocOpenMethod;
                            VAR howToTranslate: FileTranslationSpec)
                            : OSErr;
```

Translating Files

```
FUNCTION TranslateFile     (sourceDocument: FSSpec;
                            destinationDocument: FSSpec;
                            howToTranslate: FileTranslationSpec): OSErr;
```

C Summary

Constants

```
/*Gestalt selectors and response bit numbers*/
enum {
   #define gestaltTranslationAttr      'xlat'   /*Translation Manager*/
```

```
   gestaltTranslationMgrExists            = 0      /*TM is present*/
};
enum {
   #define gestaltStandardFileAttr              'stdf'   /*Std File Package*/
   gestaltStandardFile58                        = 0,
   gestaltStandardFileTranslationAware          = 1,
   gestaltStandardFileHasColorIcons             = 2
};
enum {
   #define gestaltEditionMgrAttr                'edtn'   /*Edition Manager*/
   gestaltEditionMgrPresent                     = 0,
   gestaltEditionMgrTranslationAware            = 1
};
enum {
   #define gestaltScrapMgrAttr                  'scra'   /*Scrap Manager*/
   gestaltScrapMgrTranslationAware              = 0
};

enum {domCannot, domNative, domTranslateFirst, domWildcard};
```

Data Types

```
typedef OSType             FileType;                /*file types*/
typedef ResType            ScrapType;               /*scrap types*/

typedef long               FileTranslationSpec[12];

typedef short              DocOpenMethod;
```

Translation Manager Routines

Getting Translation Information

```
pascal OSErr GetFileTypesThatAppCanNativelyOpen
                            (short appVRefNumHint, OSType appSignature,
                             FileType* nativeTypes);
pascal OSErr ExtendFileTypeList
                            (const FileType* originalTypeList,
                             short numberOriginalTypes,
                             FileType* extendedTypeList,
                             short* numberExtendedTypes);
```

```
pascal OSErr CanDocBeOpened
                            (const FSSpec* targetDocument,
                             short appVRefNumHint,
                             OSType appSignature,
                             const FileType* nativeTypes,
                             Boolean onlyNative,
                             DocOpenMethod* howToOpen,
                             FileTranslationSpec* howToTranslate);
```

Translating Files

```
pascal OSErr TranslateFile  (const FSSpec* sourceDocument,
                             const FSSpec* destinationDocument,
                             const FileTranslationSpec* howToTranslate);
```

Assembly-Language Summary

Data Structures

File Translation Specification

0	`data`	48 bytes	private data used by the Translation Manager

Trap Macros

Trap Macros Requiring Routine Selectors

`_TranslationDispatch`

Selector	Routine
$0009	`ExtendFileTypeList`
$000C	`TranslateFile`
$001C	`GetFileTypesThatAppCanNativelyOpen`
$001E	`CanDocBeOpened`

Result Codes

`noErr`	0	No error
`dirFulErr`	–33	Directory full
`dskFulErr`	–34	Not enough disk space to translate file
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`tmfoErr`	–42	Too many files open
`fnfErr`	–43	File not found
`wPrErr`	–44	Disk is write protected
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Volume is locked
`dupFNErr`	–48	Duplicate filename (rename)
`opWrErr`	–49	File already open with write permission
`paramErr`	–50	Parameter error
`extFSErr`	–58	External file system
`noTypeErr`	–102	Unrecognized file type
`memFullErr`	–108	Not enough RAM to translate file
`dirNFErr`	–120	Directory not found or incomplete pathname
`wrgVolTypErr`	–123	Volume does not support Desktop Manager
`userCanceledErr`	–128	The user canceled the translation
`invalidTranslationPathErr`	–3025	`howToTranslate` is invalid
`noTransSysInstalledErr`	–3027	No translation systems installed
`noTranslationPathErr`	–3030	Application cannot open document
`badTranslationSpecErr`	–3031	Translation path is invalid
`noPrefAppErr`	–3032	No translation preference available
`afpItemNotFound`	–5012	Could not determine kind string; or, application information not found

Summary of Translation Extensions

This section provides Pascal, C, and assembly-language summaries for the constants, data types, and routines you can use to write a translation extension. For a summary of the constants, data types, and routines that applications can use, see "Summary of the Translation Manager" beginning on page 7-63.

Pascal Summary

Constants

```
CONST
   {component flags}
   kSupportsFileTranslation              = 1;  {file translation extension}
   kSupportsScrapTranslation             = 2;  {scrap translation extension}

   {translation attributes}
   taDstDocNeedsResourceFork             = 1;  {doc needs a resource fork}
   taDstIsAppTranslation                 = 2;  {app will translate doc}

   {request codes for translation extensions}
   kTranslateGetFileTranslationList      = 0;
   kTranslateIdentifyFile                = 1;
   kTranslateTranslateFile               = 2;
   kTranslateGetScrapTranslationList     = 10;
   kTranslateIdentifyScrap               = 11;
   kTranslateTranslateScrap              = 12;
```

Data Types

```
TYPE
   FileType                = OSType;   {file types}
   ScrapType               = ResType;  {scrap types}

   TranslationAttributes   = LongInt;

   FileTypeSpec =
   RECORD
      format:           FileType;
```

```
   hint:             LongInt;
   flags:            TranslationAttributes;
   catInfoType:      OSType;
   catInfoCreator:   OSType;
END;

FileTranslationList =
RECORD
   modDate:                LongInt;
   groupCount:             LongInt;
   {group1SrcCount:        LongInt;}
   {group1SrcEntrySize:    LongInt;}
   {group1SrcTypes:        ARRAY[1..group1SrcCount] OF FileTypeSpec;}
   {group1DstCount:        LongInt;}
   {group1DstEntrySize:    LongInt;}
   {group1DstTypes:        ARRAY[1..group1DstCount] OF FileTypeSpec;}
   {repeat above six lines for a total of groupCount times}
END;
FileTranslationListPtr     = ^FileTranslationList;
FileTranslationListHandle  = ^FileTranslationListPtr;

ScrapTypeSpec =
RECORD
   format:           ScrapType;
   hint:             LongInt;
END;

ScrapTranslationList =
RECORD
   modDate:                LongInt;
   groupCount:             LongInt;
   {group1SrcCount:        LongInt;}
   {group1SrcEntrySize:    LongInt;}
   {group1SrcTypes:        ARRAY[1..group1SrcCount] OF ScrapTypeSpec;}
   {group1DstCount:        LongInt;}
   {group1DstEntrySize:    LongInt;}
   {group1DstTypes:        ARRAY[1..group1DstCount] OF ScrapTypeSpec;}
   {repeat above six lines for a total of groupCount times}
END;
ScrapTranslationListPtr    = ^ScrapTranslationList;
ScrapTranslationListHandle = ^ScrapTranslationListPtr;

TranslationRefNum          = LongInt;
```

Translation Extension Routines

Managing Translation Progress Dialog Boxes

```
FUNCTION SetTranslationAdvertisement
                            (refNum: TranslationRefNum;
                             advertisement: PicHandle): OSErr;
FUNCTION UpdateTranslationProgress
                            (refNum: TranslationRefNum;
                             percentDone: Integer;
                             VAR canceled: Boolean): OSErr;
```

Translation Extension-Defined Routines

File Translation Extension Routines

```
FUNCTION DoGetFileTranslationList
                            (self: ComponentInstance;
                             translationList: FileTranslationListHandle)
                             : ComponentResult;
FUNCTION DoIdentifyFile     (self: ComponentInstance;
                             theDoc: FSSpec;
                             VAR docKind: FileType): ComponentResult;
FUNCTION DoTranslateFile    (self: ComponentInstance;
                             refNum: TranslationRefNum;
                             srcDoc: FSSpec;
                             srcType: FileType;
                             srcTypeHint: LongInt;
                             dstDoc: FSSpec;
                             dstType: FileType;
                             dstTypeHint: LongInt): ComponentResult;
```

Scrap Translation Extension Routines

```
FUNCTION DoGetScrapTranslationList
                            (self: ComponentInstance;
                             list: ScrapTranslationListHandle)
                             : ComponentResult;
FUNCTION DoIdentifyScrap    (self: ComponentInstance;
                             dataPtr: Ptr;
                             dataLength: Size;
                             VAR dataFormat: ScrapType): ComponentResult;
```

```
FUNCTION DoTranslateScrap   (self: ComponentInstance;
                             refNum: TranslationRefNum;
                             srcDataPtr: Ptr;
                             srcDataLength: Size;
                             srcType: ScrapType;
                             srcTypeHint: LongInt;
                             dstData: Handle;
                             dstType: ScrapType;
                             dstTypeHint: LongInt): ComponentResult;
```


C Summary

Constants

```
/*component flags*/
#define kSupportsFileTranslation          1  /*file translation extension*/
#define kSupportsScrapTranslation         2  /*scrap translation extension*/

/*translation attributes*/
#define taDstDocNeedsResourceFork         1  /*doc needs a resource fork*/
#define taDstIsAppTranslation             2  /*app will translate doc*/

/*request codes for translation extensions*/
enum {
   kTranslateGetFileTranslationList       = 0,
   kTranslateIdentifyFile,
   kTranslateTranslateFile,
   kTranslateGetScrapTranslationList      = 10,
   kTranslateIdentifyScrap,
   kTranslateTranslateScrap
};
```

Data Types

```
typedef OSType             FileType;              /*file types*/
typedef ResType            ScrapType;             /*scrap types*/

typedef unsigned long      TranslationAttributes;

struct FileTypeSpec {
   FileType                   format;
   long                       hint;
```

```
   TranslationAttributes      flags;
   OSType                     catInfoType;
   OSType                     catInfoCreator;
}
typedef struct FileTypeSpec FileTypeSpec;

struct FileTranslationList {
   unsigned long              modDate;
   unsigned long              groupCount;
   /*unsigned long            group1SrcCount;*/
   /*unsigned long            group1SrcEntrySize;*/
   /*FileTypeSpec             group1SrcTypes[group1SrcCount];*/
   /*unsigned long            group1DstCount;*/
   /*unsigned long            group1DstEntrySize;*/
   /*FileTypeSpec             group1DstTypes[group1DstCount];*/
   /*repeat above six lines for a total of groupCount times*/
};
typedef struct FileTranslationList FileTranslationList;
typedef FileTranslationList   *FileTranslationListPtr,
                              **FileTranslationListHandle;

struct ScrapTypeSpec {
   ScrapType                  format;
   long                       hint;
}
typedef struct ScrapTypeSpec ScrapTypeSpec;

struct ScrapTranslationList {
   unsigned long              modDate;
   unsigned long              groupCount;
   /*unsigned long            group1SrcCount;*/
   /*unsigned long            group1SrcEntrySize;*/
   /*ScrapTypeSpec            group1SrcTypes[group1SrcCount];*/
   /*unsigned long            group1DstCount;*/
   /*unsigned long            group1DstEntrySize;*/
   /*ScrapTypeSpec            group1DstTypes[group1DstCount];*/
   /*repeat above six lines for a total of groupCount times*/
};
typedef struct ScrapTranslationList ScrapTranslationList;
typedef ScrapTranslationList  *ScrapTranslationListPtr,
                              **ScrapTranslationListHandle;

typedef long                  TranslationRefNum;
```

Translation Extension Routines

Managing Translation Progress Dialog Boxes

```
pascal OSErr SetTranslationAdvertisement
                            (TranslationRefNum refnum,
                             PicHandle advertisement);
pascal OSErr UpdateTranslationProgress
                            (TranslationRefNum refnum,
                             short percentDone,
                             Boolean* canceled);
```

Translation Extension-Defined Routines

File Translation Extension Routines

```
pascal ComponentResult DoGetFileTranslationList
                            (ComponentInstance self,
                             FileTranslationListHandle translationList);
pascal ComponentResult DoIdentifyFile
                            (ComponentInstance self,
                             const FSSpec* theDoc,
                             FileType* docKind);
pascal ComponentResult DoTranslateFile
                            (ComponentInstance self,
                             TranslationRefNum refNum,
                             const FSSpec* srcDoc,
                             FileType srcType,
                             long srcTypeHint,
                             const FSSpec* dstDoc,
                             FileType dstType,
                             long dstTypeHint);
```

Scrap Translation Extension Routines

```
pascal ComponentResult DoGetScrapTranslationList
                            (ComponentInstance self,
                             ScrapTranslationListHandle list);
pascal ComponentResult DoIdentifyScrap
                            (ComponentInstance self,
                             const void* dataPtr,
                             Size dataLength,
                             ScrapType* dataFormat);
```

```
pascal ComponentResult DoTranslateScrap
                              (ComponentInstance self,
                               TranslationRefNum refNum,
                               const void* srcDataPtr,
                               Size srcDataLength,
                               ScrapType srcType,
                               long srcTypeHint,
                               Handle dstData,
                               ScrapType dstType,
                               long dstTypeHint);
```

Assembly-Language Summary

Data Structures

File Type Specification

0	`format`	4 bytes	the file type
4	`hint`	4 bytes	reserved for use by your translation extension
8	`flags`	4 bytes	flags for controlling translation
12	`catInfoType`	4 bytes	the file's catalog type
16	`catInfoCreator`	4 bytes	the file's catalog creator

File Translation List

0	`modDate`	4 bytes	the creation date of the file translation list
4	`groupCount`	4 bytes	the number of translation groups that follow

Scrap Type Specification

0	`format`	4 bytes	the scrap type
4	`hint`	4 bytes	reserved for use by your translation extension

Scrap Translation List

0	`modDate`	4 bytes	the creation date of the scrap translation list
4	`groupCount`	4 bytes	the number of translation groups that follow

Trap Macros

Trap Macros Requiring Routine Selectors

_TranslationDispatch

Selector	Routine
$0001	UpdateTranslationProgress
$0002	SetTranslationAdvertisement

Result Codes

noErr	0	No error
dskFulErr	–34	Not enough disk space to translate file
fnfErr	–43	Document not found
paramErr	–50	Parameter error
noTypeErr	–102	Unrecognized file or scrap type
memFullErr	–108	Not enough memory
dirNFErr	–120	Source or destination directory does not exist
wrgVolTypErr	–123	Volume does not support Desktop Manager
userCanceledErr	–128	The user canceled the translation
invalidTranslationPathErr	–3025	srcType to dstType is not a valid path
couldNotParseSourceFileErr	–3026	Source document is not of type srcType
afpItemNotFound	–5012	Could not determine kind string; or, application information not found