For XTND 1.3

XTND Programmer's Guide

Developer Technical Publications

 Apple Computer, Inc.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and Tables

Chapter 1 Overview

This chapter is an overview of the XTND architecture. It summarizes the XTND document model, user interface, and programming interface. At the end of Chapter 1 is a description of the data types and conventions used throughout this document.

About XTND

XTND is an architecture that allows Macintosh® applications to read and to write files in a potentially unlimited number of file formats. The XTND architecture includes the following components:

- XTND-capable applications, which are linked with a small library of “glue” routines.
- The XTND System, a file dynamically loaded by the glue routines. The XTND System enumerates the open-ended set of file translators via the standard XTND search path by displaying a modified Standard File dialog box to select the file and format for reading or writing, loads the appropriate translator, and steps aside to allow the application to communicate directly with the translator.
- A standard text document model which specifies the “grammar” of text documents, as well as an Application Programming Interface (API) for reading and writing such standard documents. This API is used to communicate between the application and the translator.
- An open-ended set of XTND translator files stored on the Standard XTND search path. Each translator file contains resources for reading and writing one or more file formats.

The benefits of this architecture include the following:

- The process of importing and exporting foreign file formats is seamlessly integrated into the Open and Save As dialog boxes, minimizing the number of extra steps required to read and write such files.
- Users can customize their machines by installing only the required translators.
- Applications can share translators.
- XTND-capable applications and translators can be released independently, as long as compatibility rules are observed.

This document shows you how to implement XTND translator capability in your application. Although this document describes only the importing and exporting of text documents, you can use the XTND System library with other types of translators; only the communication between the application and the translator will change.

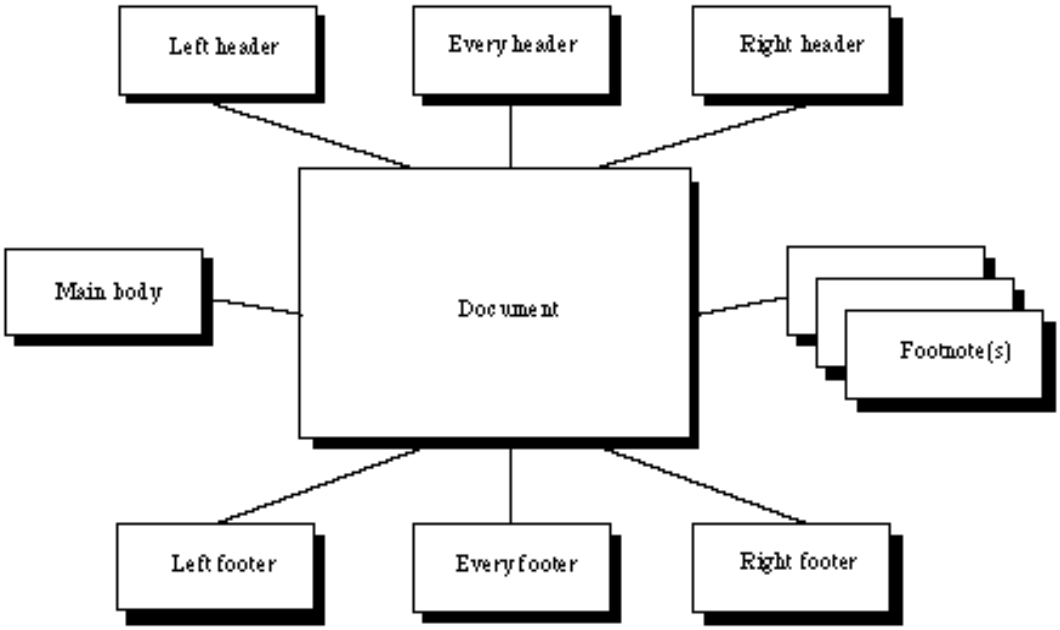
This document is a part of the XTND Developer’s Kit. The first chapter provides an overview of the XTND document model, user interface, programming interface, and programming conventions. The second chapter describes how to use XTND technology to allow your application to communicate with external code resources, such as file translators. The third chapter describes how to implement XTND-compatible file translators. The appendixes include resource templates and C header files as well as extracts from sample translators. The XTND Developer’s Kit also includes 3.5” floppy disks containing the complete header files and other components necessary to build XTND-capable applications and translators.

u *Note:* Install all XTND translators and the file named "Claris XTND System" into the Extensions folder inside your System Folder if you are using Macintosh System 7.0 or later versions. If you are using an earlier version of the Macintosh System Software, install all XTND translators and the file named "Claris XTND System" into a folder named "Claris" in your System Folder.

The XTND document model

The XTND document model describes the objects that can occur in a document, along with their attributes and sequence. In a sense, a document model is a grammar in that it guides the parsing and generation of “grammatical” documents. Thus, a document model describes the order of objects as well as their dependent relationships. Figure 1-1 illustrates the kinds of text objects you can import and export using XTND translators.

n **Figure 1-1** Objects imported and exported by XTND translators



An XTND text document consists of one or more *stories*. In Figure 1-1,*left header, every header, right header, main body, left footer, every footer, right footer,* and *footnotes* are all stories. The document has attributes, such as number of columns, the width of the gutter , the width between columns, page margins, and starting page number.

A *story* is a "flow" of paragraphs and pictures that can be "poured" into a page layout.

A *paragraph* consists of a sequence of *text runs* and optional *special characters* and a *paragraph ending character*.

A paragraph's *attributes* include line leading, leading before and after the paragraph, first line indent, left indent, right indent, and tab stops.

A *text run* consists of a string of *characters* with a homogeneous set of *font, size,* and *style* attributes. The characters are taken from the standard Macintosh character set.

A *picture* is a sequence of QuickDraw™ commands recorded in the standard Macintosh PICT format.

Special characters signify a specially-computed string value or picture to be inserted in the running flow of text. Examples include the page number, the current date, and footnote references.

A *paragraph ending character* represents an instruction to the text-flow process and is placed in the running flow of text. Examples include page breaks, column breaks, and hard return characters.

The XTND User Interface

The user installs XTND translators by dragging translator icons into any of the following folders, or any subfolders within them:

- the Extensions Folder (System 7 only) (Recommended)
- the Claris folder (located in the System Folder)

An example of the icon design for a typical XTND translator is shown below:

n **Figure 1-2** Icon for an XTND translator

In addition, the user must install the file named *Claris XTND System* somewhere in the XTND search path, which is as follows:

- the Extensions Folder (System 7 only)
- the System Folder
- the Claris folder (located in the System Folder)

Figure 1-3 shows the XTND System icon.

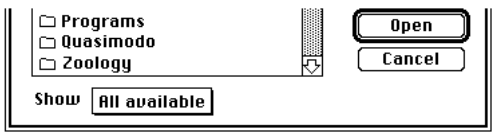
n **Figure 1-3** XTND System icon

The *XTND System* implements the standard XTND user interface for specifying a file to be opened or saved. It allows the user to choose the format of the files shown in the Open dialog box and to specify the format of a file in the Save As dialog box. Other functions allow an application to request a list of translators or determine the availability of a translator that can open a specified file.

XTND applications use these packages in place of those in the Macintosh Standard File package. When the user selects the Open or Save As menu commands, the application calls this library package to find or name the document.

In the standard Open dialog box, the library places a pop-up menu below the file list, as in Figure 1-4.

n **Figure 1-4** A list of files in an Open dialog box

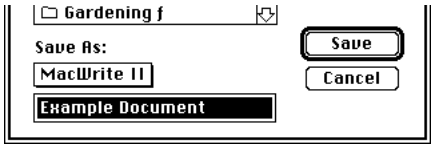


When the default option (“All available”) is in effect, all available files that can be opened are displayed in the file list. If the user chooses one of the pop-up menu selections, then only files that match that file specification are displayed.

Some applications can read and write certain file formats without using external XTND translators. These file formats are listed above a dotted line in the pop-up menu listing of translator file formats. Although these formats appear to the user to behave just like XTND translators, they are built into the application and need not be installed.

During a Save As command, the pop-up menu is similarly located, between the file list and the document name. Figure 1-5 shows an example.

Figure 1-5 A pop-up menu in a Save As dialog box



After identifying the file and file format, the user clicks the Open (or Save) button to initiate the Open (or Save). The appropriate XTND translator is automatically loaded to import (or export) the file.

Applications can choose to override this user interface by supplying their own import and export dialog boxes. These applications can use the XTND System to generate the list of available translators and use the translators to read or write documents.

The *XTND System* creates a file named *XTND Translator List* which contains the list of available translators. This list is updated whenever the XTND System detects a change in the installed translators. The XTND System checks the modification dates of the following folders to determine if the installed translators have changed:

- a folder named *Claris Translators* in the application folder
- a folder named *Claris Translators* in the *System Folder*
- a folder named *Claris Translators* in the *Claris* folder (the Claris folder is within the System Folder)

Because translators may be installed in folders other than these, however, sometimes the list is not updated. To force an update of the list of translators, the user can remove the *XTND Translator List* file. In response, the XTND System creates a new list file in one of the following locations:

- the *Preferences Folder* (System 7 only);

- the *Claris* folder (located in the *System Folder*);
- the *System Folder* (if no *Claris* folder exists in the *System Folder*);

Figure 1-6 shows the XTND Translator List icon.

- **Figure 1-6** XTND Translator List icon

XTND Translator List

Important Your End User documentation should include the following statement: "Install all translators and the file named Claris XTND System into the Extensions folder when using System Software version 7.0. If you are using an earlier version of System Software, install all translators and the file named Claris XTND System into the Claris folder located in the System folder."

The XTND Programming Interface

The XTND library is stored in a file named *Claris XTND System*, which must be located somewhere in the standard XTND search path described earlier in this chapter. You will find Claris *XTND System* on the accompanying 3.5-inch disk. It should be placed (along with the XTND translators) in the *Claris* folder in the user's System Folder, and it can be placed in the system folder if the system has no *Claris* Folder. This library contains the routines that locate,organize, and select from among the available translators. In this way, the application need not implement these routines, and you can update the routines without affecting applications that use the library.

The `XTNDInterface.Lib.o` and `XTNDInterface.π` libraries(also on the accompanying 3.5-inch disk) are for MPW® and THINK C™, respectively. These libraries allow your application to call the XTND library. They contain a procedure to initialize the library, the `XTNDGetFile` and `XTNDPutFile` procedures, other utility procedures, and, finally, the procedure to dispose of the library when the application is finished with it. Prototypes for all XTND library routines, along with other constants and data structures needed in XTND, are in the file `XTNDInterface.h` if you are using C. `XTNDInterFace.p` contains the corresponding Pascal definitions. To use the XTND System, you must link the appropriate library to your application.

u *Note:* When creating an XTND-capable application, be sure to take into account that memory is required to load the XTND System and to perform XTND functions.

The initialization routine, `XTNDInitTranslators`, attempts to create a handle to some global memory, then to search for the *Claris XTND System* file, and finally to locate and create a list of available XTND translators. If the attempt to locate the *Claris XTND System* file or to create the global handle space fails, the initialization routine returns an error. None of the other XTND routines (except `XTNDCloseTranslators`) should be called if an error occurs in initialization.

The `XTNDGetFile` routine allows the user to specify a file and an XTND translator to be used for opening the file. The application specifies the criteria that determine which XTND translators are available, optionally specifies a dialog template, filter procedure, and dialog hook; and then calls `XTNDGetFile`, which returns a regular `SFReply` record and a pointer to the selected XTND translator.

The XTNDPutFile routine allows the user to specify a file to be saved and an XTND translator to be used for writing to the file. As with the XTNDGetFile routine, the application calls XTNDPutFile, which returns a standard SFReply record and a pointer to the selected XTND translator.

The XTNDCloseTranslators routine tidies up for the library. The application calls XTNDCloseTranslators, which unlocks and releases all handles and pointers and removes the library segments and resources that were loaded into memory. It also disposes of the global handle that the application created. You may call this routine even if the initialization was not completed successfully, but there is no need to do so.

The XTNDMatchFile routine finds and returns a translator that opens a specified file, or returns a result code indicating that no suitable translator could be found.

The XTNDSelectTranslator routine returns a list of translators that meet the criteria specified by the application. This is how the application finds out which translators are available to perform different tasks.

The XTNDRebuildTransList routine updates the list of available translators while the application is running. This routine provides a convenient alternative to quitting the application, discarding the XTND Translator List file, and restarting the application. The update routine is useful when translators are installed while the application is running.

The XTND 1.3 Developer's Kit includes a 3.5 ” disk with sample applications. XTEStyleSample (a Pascal example) and TESample (a C example) are sample programs that illustrate the methods outlined here.

Data Types and Conventions

This section defines the standard data types and conventions used in the XTND architecture.

General Information

Unless otherwise noted, the following apply:

- TRUE is 1 (0x01) and FALSE is 0 (0x00).
- All numerical fields are signed short integers (16 bits).
- All measurements are stored in fixed fields as points (that is, 1 inch is stored as 0x00480000).

- Bit 7 is always the *most significant* bit in a byte, and bit 0 is always the *least significant,* as shown in Figure 1-7.

- **Figure 1-7** Bit 7 is the most significant

Reserved Characters

The characters in Table 1-1 have special meaning in an XTND text document. Characters in italics are considered special characters; paragraph ending characters are indicated by boldface.

- **Table 1-1** Reserved characters

Character	Value	Description
2	*(0x02)*	*Page number*
3	*(0x03)*	*Footnote reference (within a footnote)*
4	*(0x04)*	*Picture*
5	*(0x05)*	*Footnote reference (within the main body)*
6	**(0x06)**	**Merge break character**
7	(0x07)	Hard return (return within a paragraph)
9	(0x09)	Tab
11	**(0x0B)**	**Column break**
12	**(0x0C)**	**Page break**
13	**(0x0D)**	**Return (paragraph break)**
21	*(0x15)*	*Short date*
22	*(0x16)*	*Abbreviated date*
23	*(0x17)*	*Long date*
24	*(0x18)*	*Abbreviated with day date*
25	*(0x19)*	*Long with day date*
26	*(0x1A)*	*Time*

31	(0x1F)	Discretionary hyphen

Paragraph attributes

The application creates the paragraph format record (Num`ParaFmts`); the translator reads (but should not modify) the elements. Table 1-2 shows the structure of the record.

Table 1-2 Structure of the `NumParaFmts` record

Length	Description
4 bytes	(Fixed) Offset in points of the left indent relative to the edge of the left column
4 bytes	(Fixed) Offset in points of the first line indent relative to the left indent
4 bytes	(Fixed) Offset in points of the right indent from the edge of the right column
4 bytes	Leading. This field can have different meanings, according to the value of the leading units field, described later. (See the next section, “Units of measurement,” for a description of the possible values for this field.)
4 bytes	Space before. Extra space before the current paragraph. This field can take on different meanings based on the space before units field described later. (See the next section, “Units of measurement,” for a description of the possible values for this field.)
4 bytes	Space after. Extra space after the current paragraph. This field can have different meanings based on the space after units field as described later. (See the next section, “Units of measurement,” for a description of the possible values for this field.)
4 bytes	Leading units. (See the next section, “Units of measurement,” for a description of the possible values for this field.)
4 bytes	Space before units. (See the next section, “Units of measurement,” for a description of the possible values for this field.)
4 bytes	Space after units. (See the next section, “Units of measurement,” for a description of the possible values for this field.)

Units of measurement

The Leading, Space before, and Space after fields (called size fields) are each associated with a corresponding units field in a paragraph record. A units field determines the unit of measurement (such as lines or inches) the user has specified for the corresponding size field. A size field contains either a fixed-point value (which always contains the line spacing in points) or a fixed-line spacing value (where 0 means 1 line, 1 (0x00010000) means 2 lines, and so on), based on the value in the units field. Table 1-3 shows the possible values for the units field and the corresponding type of entry in the size field, and Table 1-4 shows an example.

n **Table 1-3** Units field values and corresponding size field types

Units Field Value		Description	Type of Entry In the Size Field
-1	Lines	Fixed-line spacing	
0	Points	Fixed-point value	
2	Millimeters	Fixed-point value	
3	Inches	Fixed-point value	
4	Centimeters	Fixed-point value	

n **Table 1-4** Sample units field values and corresponding size field types

What the User Sees	Units Value	Size Value
1 Inch	3	72 (0x00480000)
36 Points	0	36 (0x00240000)
1 Line	-1	0
2.5 Lines	-1	1.5 (0x00018000)

Tabs

A tab array is comprised of 20 tabs, each of which is defined in Table 1-5.

n **Table 1-5** Tab definitions

Length	Description
1 byte	Justification: Value — Meaning 0 — Left 1 — Center 2 — Right 3 — Character-aligned
1 byte	(Character) Fill character.
4 bytes	(Fixed) Offset in points from the edge of the left column. A value of -1 means that there are no more tabs.
1 byte	(Character) If this is a character-aligned tab, the character on which the tab is aligned.
1 byte	unused; should be set to 0.

Chapter 2 Writing XTND-Capable Applications

Chapter 2 describes the process of using the XTND System from the application's point of view. This process includes:

- locating and loading the XTND library
- calling the library to allow the user to select the file and translator
- loading the appropriate translator and opening the data file
- calling the translator repeatedly to read (or write) objects
- calling the translator to clean up and terminate

This process can be customized in several ways. Some examples follow:

- The application can handle "built-in" file formats internally without using external XTND translators.
- The application can specify its own GetFile (or PutFile) dialog box , use the XTND System to enumerate translators, and use a translator to read and write file formats.
- The application can instruct the library to include or exclude translators according to specific criteria.
- The application can request a list of translators, process it, and ask the XTND System to display the modified list.
- Applications and translators can be localized into other languages.

Locating the XTND library

`XTNDInitTranslators` is the initialization portion of the XTND Standard File Package.

```
FUNCTION XTNDInitTranslators(transVersion : INTEGER; xtndSystemName,
                                clarisFolderName : Str255) : OSErr;
```

`XTNDInitTranslators` searches through the folders as previously specified and locates a file named *Claris XTND System*. The application must pass in the strings that name the XTND System file and the Claris folder. The Claris folder is named "Claris". If you don't wish to use the Claris folder, you can omit this parameter by passing either nil or an empty string. You should store these strings in resources in the application so that they can be localized. (See the sample applications on the 3.5-inch disk provided with the XTND 1.3 Developer's Kit.) Upon finding this file, the function opens it, reads some resources, and checks their validity. If the version of the library is not compatible with the library used by the application, `badXTNDVersionErr` is returned. Because new features are added to the library periodically, applications using the new features will not work correctly with older versions of the library. Such applications should always be shipped with the current library and with instructions to discard any older versions of the library.

If no file is found, then an error number is returned; if all is well, `noERR` is returned. In versions of XTND prior to 1.3, the `XTNDGetFile` or `XTNDPutFile` procedure contained a default routine that was executed when it was called after XTND failed to initialize. This default routine was in `CLSFLibrary.c`, which is no longer supplied. In the current version of XTND, the application must check whether XTND was initialized before calling other XTND routines and must take appropriate action if it was not.

After locating the library file, `XTNDInitTranslators` searches for any XTND translators that the user has installed. It searches for the `XTND Translator List` file (see Chapter 1 for the search path for this file) and creates the file if it does not exist. It then checks whether the list of translators in this file needs updating. If the list is current, then the XTND library builds a table in memory; this table contains the location of and ‘FTYP’ information about the available translators. (See the section “The FTYP Resource” in Chapter 3 for a description of an 'FTYP' resource.)

If the list of translators in the `XTND Translator List` file is not current, then the XTND library rebuilds the translator list by searching for the installed translators. First, if System 7 is running, it searches the Extensions folder; next, it searches for a folder named *Claris Translators* in the application folder; next, it searches for a folder named *Claris Translators* in the System Folder; and then it searches for a folder named *Claris* in the System Folder. The library searches any folder it finds for translators. The translators need not be at the root level in any of these folders because the library searches all subfolders.

Any translators whose version number matches the value in the `transVersion` parameter is collected by the library and made available to the application. Current translators are all version 2. The version number of the translators will not change unless the interface to the translators changes. For this reason, translators whose version numbers differ from that of the version of XTND in use are incompatible with that version of XTND.

Importing a file

The `XTNDGetFile` routine presents a dialog box listing the names of a group of files from which the user can select one to open.

```
FUNCTION XTNDGetFile(ParamPtr : SFParamPtr) : BOOLEAN;
```

Like `SFGetFile`, `XTNDGetFile` repeatedly gets and handles events until the user selects a file that can be opened or clicks Cancel. The routine reports the user's selection via the Standard File Package's reply record and through the `SFParamBlock`'s `ioResult`. The application can specify its own Get File dialog box to be used and can specify a modal dialog filter proc and a SF dialog hook (actually an `XTNDDialogHook`) to be used with its dialog.

`XTNDGetFile` returns the Boolean variable taken from the user's reply record. Thus, the application can easily check to see whether a file was actually selected.

Implementation of `XTNDGetFile`

The first step is to set up the Standards array. This array allows the application to inform the XTND library of the "built-in" file formats that are handled directly by the application. These internal file formats are listed in the pop-up menu between "All available" and the separating dotted line. All external XTND translator names appear below the dotted line.

The `TransDescribe` block is defined in `XTNDInterface.h`. See Table 2-1.

Table 2-1 `TransDescribe` block

Field Name	Size	Default Value	Description
`version`	short	2	Low byte contains current version of translators (in low 7 bits) and flag (in high bit) which is TRUE if translator can only be used by applications that are in the same language. High byte contains language of translator.
`translatorType`	ResType		Translator type / Current usage 'FLTI', 'FLTE' — TEXT Import, Export 'PFLT' — PICT Import
`codeResID`	short	0	Used by `TransDescribe` to store the resource ID of a translator.
`FDIFResID`	short	-1	Indicates that there is no need to check for an 'FDIF', if this is an import translator.
		OR	
		-2	Indicates that this is an export translator.
`numVersBytes`	short	0	Number of version bytes to look for (used in IMPORT only).
`versBytesOffset`	long		Offset of version bytes from beginning of file (used in IMPORT only).
`versBytes`	char[16]		Version bytes to look for (used in IMPORT only).
`appWDRefNum`	short	0	Internal XTND use; must be zero.
`unused1`	short	0	Internal XTND use; must be zero.
`pathLength`	short	0	Internal XTND use; must be zero.
`flags`	long	0	Translator characteristics.
`transIndex`	short	0	Internal XTND use; must be zero.
`resRefNum`	short		Reference number of the resource fork of the translator, if open; 0 if not open.
`directoryID`	long		Directory containing this translator.
`vRefNum`	short		Volume containing this translator.
`fileName`	char[32]		Name of this translator file.

(Continued)

n **Table 2-1** `TransDescribe` block (Continued)

Field Name	Size	Default Value	Description
`numMatches`	short		Number of file type and creator matches accepted for this file type.
`matches`	MatchInfo[ ]		An array of `MatchInfos`, one for each match in `numMatches`.
`docCreator`	OSType		Creator ID for this match.
`docType`	OSType		Type ID for this match.
`exactMatch`	char	TRUE:	`docCreator` AND `docType` must match.
		FALSE:	Only `docType` must match.
`creatorAndTypeMask`	char	0	Ignore certain characters in match.
`name`	char[32]		Name to appear in pop-up menu.

For example, this is how MacWrite II might set up the translator descriptions in the `Standards` array.

```
for (loop = 0; loop < NUMSTDS; loop++)
{
      Standards[loop].version = 2;
      Standards[loop].translatorType = 'FLTI';
      Standards[loop].codeResID = 0;
      Standards[loop].FDIFResID = -1;
      Standards[loop].numVersBytes = 0;
      Standards[loop].pathLength = 0;
      Standards[loop].flags = 0;
      Standards[loop].numMatches = 1;
      Standards[loop].matches[0].docCreator = 'MWII';
      Standards[loop].matches[0].docType = (loop == 0 ? 'MW2D':'MW2S');
      Standards[loop].matches[0].exactMatch = FALSE;
      Standards[loop].matches[0].creatorAndTypeMask = 0;
}
Standards[2].matches[0].docType = 'TEXT';
Standards[2].matches[0].exactMatch = FALSE;
/* Get the names of each translator */
GetIndString(Standards[0].name, STRINGS, 1); /* MacWrite II */
GetIndString(Standards[1].name, STRINGS, 2); /* MacWrite II Stationery */
GetIndString(Standards[2].name, STRINGS, 3); /* Text */
```

You can also use this array in `XTNDPutFile`, unless you write different types than you read.

After setting up the Standards array, you set up an SFParamBlock, as shown in Table 2-2, specifying the types of files from which the user can choose.

n **Table 2-2** SFParamBlock

In/Out (->/<-)	Parameter	Description
-->	allowFlags	Which translators are allowed to be used.
	allowText	Allows text translators.
	allowGraphics	Allows graphic translators (future implementation).
	allowDataBase	Allows database translator (future implementation).
	allowPict	Allows PICT translators.
	allowOtherTypes	Allows other translator types (future implementation).
	allowAllTranslators	Allows all translator types or any combination (for example, allowAllTranslators - allowPict = everything except PICT translators).
-->	numStandard	The number of standard transDescribe.
-->	standard	A pointer to the Standard array of TransDescribe.
<--	ioResult	Any error that might occur during a call.
<->	chosenTranslator	On input, the number of translators in the list being supplied by the application (optional, used in conjunction with useTransList); on output, the number of the selected translator.
<->	theChosenTranslator	On input, a pointer to a list of translators being supplied by the application (optional, used in conjunction with useTransList); on output, pointer to a TransDescribe record that specifies which translator was selected to open the specified file.
<--	fileReply	A pointer to an SFReply record.

(Continued)

n **Table 2-2** `SFParamBlock` (Continued)

In/Out (->/<-)	Parameter	Description
-->	`applicNativeType`	Unused in `XTNDGetFile`.
-->	`XTNDDlogHook`	Pointer to an SF dialog hook that handles your items in the dialog box (optional).
<->	`currentMenuItem`	The item chosen in the pop-up menu on a previous call to `GetFile`.
<->	`currentSaveItem`	Unused in `GetFile`.
-->	`where`	A point in normal Macintosh coordinates where you wish the dialog box to appear. If you choose (0,0) it will be placed one-third of the way down the main screen and one-half of the way across the screen.
-->	`prompt`	A Pascal string that is written at the top of the Get File dialog box, above the file list.
-->	`buttonTitle`	A Pascal string to replace the name of the Open button title. If left NULL, “Open” is used as the name.
-->	`origName`	Unused during a `GetFile` call.
-->	`dialogID`	Resource number of your `GetFile` dialog template (optional).
-->	`SFFilterProc`	Pointer to your modal dialog filter proc (optional).
-->	`showAllFiles`	TRUE if you want all files shown in the file list; FALSE otherwise.
-->	`useTransList`	TRUE if you are supplying a list of translators (in `theChosenTranslator`, with the count in `chosenTranslator`) that is to be used to build the pop-up menu; otherwise FALSE.

u *Note:* Any unused fields in this or other XTND parameter blocks should be set to 0, even if they are currently unused. They may be used in the future, and leaving random values in these fields will cause your application to break when the fields are used, even though it works correctly now.

The allowFlags field specifies the type and use of translators that you wish to include in the pop-up menu. The GetFile routine will automatically set the usage to allowImport, so you need specify only the translator types you wish. For word processors, you would typically set this field to allowText, or allowText + allowPict, if you can import QuickDraw Pictures.

Make the standard field a pointer to the transDescribe list describing your standard types, and set the numStandard field to the number of standards in your list.

You should set currentMenuItem to some reasonable value, usually 1 for the first call to GetFile ("All available"). After this, current MenuItem can usually be ignored, since the call to GetFile sets currentMenuItem to the number of the last selected translator, and this is a good starting value for the next GetFile call. If you change the list of translators, you should probably set currentMenuItem back to 1, since the previous value may no longer be meaningful.

Set where to the point where you wish the top-left corner of the dialog box to appear. If you set it to (0, 0), the dialog box is centered on the screen.

Make prompt a pointer to a string you want to put at the top of the Get File dialog box. This string generally prompts the user to select a file. "Please select a file" is used if you set this pointer to nil. Similarly, make buttonTitle a pointer to a string you want to appear in the Open button. The name "Open" is used if you set this pointer to nil.

Usually, you set useTransList to FALSE. If you wish to bypass the standard selection process for translators and to control it yourself, you can use XTNDSelectTranslators to request a list of translators, process it, and return a list of the translators you wish to include in the pop-up menu. To do this, set useTransList to TRUE, make the theChosenTranslator field point to your list of translators (array of transDescribe), and set chosenTranslator to the number of translators in your list. If you do this, these are the only translators (besides the standards) to appear in the pop-up menu.

u *Note:* No validity checking is done on these translators, and you should set useTransList to TRUE only if you need to have total control over which translators appear in the pop-up menu. The same procedure is used to insert these translators in the menu, however, so all of the normal duplicate rejection, language checking, and sorting is still done on them.

If you wish to use your own Get File dialog box in place of the XTNDGetFile dialog box (which is used in place of the standard Get File dialog box), create your dialog box template resource and include it in your application. Your dialog box must include all of the dialog items that are in the XTND GetFile dialog box. You may add your own items following these, and if you wish to hide the existing items, you can move them offscreen (needless to say, you do this at your own risk), but they will still be active. To use your dialog box, put the resource ID of your dialog box template in the `dialogID` filed. You may also special a `ModalDialogFilterProc` if you wish to process events that are not currently processed in the dialog box. XTND calls your filter procedures after it has processed its own events, and Standard File calls the XTND filter procedure after it processes all Standard File events. Last, you may specify an `XTNDDialogHook`, which is like a `SFDialogHook`, only better.

The `XTNDDlgHook` is actually an `SFDlgHook`; however, three parameters have been have added to this function to allow it to communicate more easily with the XTND System:

```
Function MyXTNDDlgHook  (  item:integer;
                           theDialog:DialogPtr;
                           theSFParams:Ptr;
                           VAR changedFlag:boolean;
                           reserved:longint  ) : Integer;
```

- The first additional parameter is a pointer to your `SFParamBlock`. You may change the fields that affect translator selection, and the pop-up menu is rebuilt using the new parameters. Currently, the only fields used in this process are `allowFlags`, `showAllFiles`, `currentMenuItem` (or `currentSaveItem`), and `useTransList` (along with `chosenTranslator` and `theChosenTranslator`). You should not change the selection parameters when you receive an item number of -1 because this call initially draws the dialog box and your changes in this case are ignored.
- The second additional parameter (`changedFlag`) is a pointer to a Boolean variable used to indicate that one or more of the fields in the `SFParamBlock` has been changed and that the pop-up menu is to be rebuilt. This field is set to FALSE when your dialog hook is called, and you must change the value to TRUE if you wish to rebuild the menu.

u *Note:* If you change `allowFlags` or any other parameter which could change the number of translators in the menu, `CurrentMenuItem` (or `currentSaveItem`) may no longer be valid. For this reason, you may want to set `allowFlags` to 1 when you change the number of items in the menu. The list of files is rebuilt whenever the list of translators changes.

- The third parameter is a longint and is currently unused
- The remaining parameters, `item` and `theDialog`, are the same as the Standard File `SFDialogHook` parameters.

- *Note:* XTND uses the pre-System 7 Standard File dialog hook. The System 7 hook has a different parameter list and cannot directly communicate with the XTND interfaces. For more information, see Inside Macintosh, Volumes I and VI.

When using your own dialog box, you should start with the XTND standard file dialog boxes 24000 and 24001 from the Claris XTND System resource fork (get them by using ResEdit or DeRez). Add your additional items onto the ends of the dialog boxes' DITLs. Do not remove any of the items or renumber them from the original template, since XTND expects certain items with certain ID numbers. You are free to resize and to relocate items in the dialog box.

These parameters allow you to change the list of currently available translators while the dialog box is up. If other methods of selecting translators are implemented, they are accessible using the `SFParamBlock` and should be available to your dialog hook. Aside from the additional parameters, your dialog hook should work as a standard dialog hook would. XTND calls your dialog hook before the XTND dialog hook and passes items to it as to a standard dialog hook. You need not return any special item number when you change the selection parameters; XTND does not check the returned item number when the value of `changedFlag` is TRUE. When the parameters have not been changed, you should return items as you would with a standard `SFDlgHook`.

Now call `XTNDGeFile(&SFParams)`, in which `SFParams` is your parameter block.

Using the XTND Translator to read in the file

After the user has selected the file format (and `XTNDGetFile` has returned a TRUE result), then the application needs to read that file into a document. If the selected translator is one of the “standard” items, then your application can handle it as usual. If not, you need to use the XTND Translator to access the contents of the file.

To open the file, the `XTNDGetFile` procedure returns a pointer to the `TransDescribe` record of the selected translator.

After receiving the translator, initialize the import parameter block. In the following example, assume that `gImportPB` is an `ImportParmBlock` as defined in `XTNDTextTranslator.h` and that `MinusOne` is a point declared as coordinates (-1, -1). You should also assume that the variables `ParaFmts` and `tabs` have been allocated and set up in accordance with the conventions given in “Data Types and Conventions” in Chapter 1. Because the translator modifies these arrays directly, you must allocate them correctly in your application. The `markerString` parameter is a Pascal string no more than 10 characters long. The `buffer` parameter is a pointer to an array of 256 characters.

Here are the initial settings of the parameter block:

```
gImportPB.textBuffer          = buffer;
gImportPB.result              = noErr;
gImportPB.textLength          = 0;
gImportPB.txtFace             = face;
gImportPB.txtSize             = 0;
gImportPB.txtFont             = font;
gImportPB.txtColor            = color;
gImportPB.txtJust             = textLeft;
gImportPB.paraFmts            = ParaFmts;
gImportPB.tabs                = tabs;
gImportPB.numCols             = 1;
gImportPB.currentStory        = mainStory;
gImportPB.miscData            = 0;
gImportPB.storyHeight         = 0;
gImportPB.decimalChar         = '.';
gImportPB.autoHyphenate       = TRUE;
gImportPB.printRecord         = NULL;
gImportPB.startPageNum        = 1;
gImportPB.startFootnoteNum    = 1;
gImportPB.footnoteText        = markerString;
gImportPB.footnoteText[0]     = 0;
gImportPB.rulerShowing        = 2;
gImportPB.doubleSided         = FALSE;
gImportPB.titlePage           = FALSE;
gImportPB.endnotes            = FALSE;
gImportPB.showInvisibles      = FALSE;
gImportPB.showPageGuides      = TRUE;
gImportPB.showPictures        = TRUE;
gImportPB.autoFootnotes       = TRUE;
gImportPB.pagePoint           = MinusOne;
gImportPB.datePoint           = MinusOne;
gImportPB.timePoint           = MinusOne;
gImportPB.smartQuotes         = TRUE;
gImportPB.fractCharWidths     = FALSE;
gImportPB.hRes                = 72;
gImportPB.vRes                = 72;
gImportPB.theReply            = theReply;
```

Now open the XTND Translator and load it into memory. XTND defines `gImportTranslator` as a pointer to a C routine returning void, and `pChosenOne` as a pointer to the selected translator.

```
TransProcPtr gImportTranslator;
TransDescrPtr pChosenOne = gImportPB.theChosenTranslator;
   if (fserr = XTNDLoadTranslator(pChosenOne, &gImportTranslator)) {
         AlertUser("\pError while trying to load translator resource");
         return;
```

```
   }
```

When you have loaded the XTND Translator without error, open the selected document . Check to see whether the document has a resource fork. If it does, then you will open the resource fork and call the translator.

```
if (OpenRFPerm(gtheReply.fileName, gtheReply.vRefNum, fsRdPerm) == -1)
{
   if (ResError() != eofErr)
   {
        AlertUser("\pError while opening import file resources");
        XTNDReleaseTranslator(pChosenOne);
        return;
   }
   /* there is no resource fork for this file.  Set the current
            resource file to the translator we found instead. */
   UseResFile(pChosenOne->resRefNum);
}
else
{
   resfnum = CurResFile();
   gImportPB.refNum = resfnum;
   gImportPB.directive = importGetResources;
   (*gImportTranslator)(&gImportPB);
   if (gImportPB.result)
   {
        AlertUser("\pTranslator was unable to read resources");
        CloseResFile(resfnum);
        XTNDReleaseTranslator(pChosenOne);
        return;
   }
}
```

You are now ready to initialize the document-reading process. We define `ApplePB` as a `ParamBlockRec`.

```
/* Open the file read only */
fserr = 0;
ApplePB.ioParam.ioNamePtr = gtheReply.fileName;
ApplePB.ioParam.ioVRefNum = gtheReply.vRefNum;
ApplePB.ioParam.ioVersNum = 1;
ApplePB.ioParam.ioPermssn = fsRdPerm;
ApplePB.ioParam.ioMisc = 0L;

fserr = PBOpen(&ApplePB, FALSE);
if (fserr)
{
   AlertUser("\pUnable to open import file");
   if (resfnum) CloseResFile(resfnum);
```

```
   XTNDReleaseTranslator(pChosenOne);
   return;
}
fnum = ApplePB.ioParam.ioRefNum;
/* let translator do general initialization */
gImportPB.refNum = fnum;
gImportPB.directive = importInitAll;
(*gImportTranslator)(&gImportPB);
/* After completing the initialization, check for an error. */
/* If none, proceed. */
```

In some applications, you may wish to distinguish between opening a text file and inserting one. For example, when MacWrite II calls the translator in response to the `Open` routine, the import process includes headers and footers; but if it is calling the translator in response to the `Insert` routine, only the main story and associated footnotes are read in.

```
if (fromOpen)
{
   gImportPB.directive = importInitRightHeader;
   gImportPB.currentStory = rightHeaderStory;
}
else
{
   gImportPB.directive = importInitMain;
   gImportPB.currentStory = mainStory;
}
getnextstory(&gImportPB);
```

The routine `getnextstory` calls the import translator until a story is ready for import. After the call, `gImportPB` is set up so you know what to import.

```
/*****************************************************************
*     getnextstory                                               *
*                                                                *
*     Gets next story (header,footer,etc.) from translator       *
*     Input:      None.                                          *
*     Output:     None.                                          *
*     Changes:    None.                                          *
*     Effects:    None.                                          *
*****************************************************************/
static void getnextstory(thePtr)
register ImportParmBlkPtr thePtr;
{
   while (thePtr->directive != importAcknowledge)
   {
        (*gImportTranslator)(thePtr);
        switch(thePtr->directive)
```

```
        {
            default:
                thePtr->directive = importAcknowledge;
                break;
            case importInitRightHeader:
                thePtr->currentStory = leftHeaderStory;
                thePtr->directive = importInitLeftHeader;
                break;

            case importInitLeftHeader:
                thePtr->currentStory = headerStory;
                thePtr->directive = importInitHeader;
                break;

            case importInitHeader:
                thePtr->currentStory = rightFooterStory;
                thePtr->directive = importInitRightFooter;
                break;

            case importInitRightFooter:
                thePtr->currentStory = leftFooterStory;
                thePtr->directive = importInitLeftFooter;
                break;

            case importInitLeftFooter:
                thePtr->currentStory = footerStory;
                thePtr->directive = importInitFooter;
                break;

            case importInitFooter:
                thePtr->currentStory = mainStory;
                thePtr->directive = importInitMain;
                break;
        }
    }
}
```

The application needs to set up the part of its document corresponding to the story the translator wishes to return. After this, the application reads in the story by repeatedly calling the translator with the `importGetText` directive:

```
/* Reading the TEXT body of the current story */
gImportPB.directive = importGetText;
gImportPB.miscData = 0;
(*gImportTranslator)(&gImportPB);
```

Upon return, the translator has filled in the following fields:

- `result` (error number)
- `textLength` (size of text imported)
- `txtSize` (size of font)
- `txtFace` (style of character run)
- `txtFont` (font family ID)
- `txtColor` (XTND 1.3 color ID)
- `txtJust` (XTND 1.3 justification ID)
- `miscData` (used if special characters returned)
- `paraFmts` (paragraph array)
- `tabs` (tab information array)
- `textBuffer` (actual text returned)

If an error is returned at this time, you need to jump directly to closing the translator (by calling it with an `importCloseAll` directive) and disposing of the resources in use.

If `ImportFile.directive` is now set to `importAcknowledge` and the value of `textLength` is less than or equal to 0, then this story has been completed, and the appropriate close story call should be issued. (Instructions for closing a story appear later in this document).

If `directive` is not set to `importAcknowledge` or the value of `textLength` is greater than 0, and if no error has occurred, begin inserting the text run into your document.

If the value of `textLength` is 1 (that is, only one character has been imported), you need to see if it is a footnote or other special character (such as date or time). If it is a special character, please refer to the section “Reserved characters” in Chapter 1 to review how the translator returns the information.

After inserting the text, check the “floating” characters that have been imported. These characters are imported mostly from MacWrite 5.0, but other applications may use them also. If the value of datePoint, pagePoint, or timePoint is greater than or equal to 0, then the translator is returning the offset from the left margin (in points) of that special character. It is up to your application to place these “floating” characters to best advantage. (Remember to reset the point values to -1 after you have placed the character.)

After the translator returns an “end-of-story” condition (by setting `directive` to `importAcknowledge` and making the value of `textLength` less than or equal to 0), the application must close this story in the translator and open the next one.

```
if (gImportPB.directive == importAcknowledge && gImportPB.textLength
   <= 0)
{
        switch (gImportPB.currentStory)
```

```
        {
        case mainStory:
             gImportPB.directive = importCloseMain;
             (*gImportTranslator)(&gImportPB);
             if (!fserr)
             {
                  gImportPB.currentStory = footnoteStory;
                  gImportPB.directive = importInitFootnote;
                  (*gImportTranslator)(&gImportPB);
             }
        break;
        case footnoteStory:
             gImportPB.directive = importCloseFootnote;
             (*gImportTranslator)(&gImportPB);
             if (!fserr)
             {
                  gImportPB.directive = importInitFootnote;
                  (*gImportTranslator)(&gImportPB);
             }
        break;

        case rightHeaderStory:
             gImportPB.directive = importCloseRightHeader;
             (*gImportTranslator)(&gImportPB);
             if (!fserr)
             {
                  gImportPB.currentStory = leftHeaderStory;
                  gImportPB.directive = importInitLeftHeader;
                  getnextstory(&gImportPB);
             }
        break;
        case leftHeaderStory:
             etc.…
        }
}
```

When all of the footnotes have been read, you must close the translator correctly. Begin by allowing the translator to clean up all its variables and dispose of its resources. This is done with a call to `importCloseAll`. After this, you can dispose of the translator resource itself. If an error occurs during the importing of a document, you should jump directly to this portion of the code.

```
gImportPB.directive = importCloseAll;
(*gImportTranslator) (&gImportPB);
if (resfnum) CloseResFile(resfnum);
FSClose(fnum);
XTNDReleaseTranslator(pChosenOne);
```

Saving a file

```
FUNCTION XTNDPutFile(paramPtr: SFParamPtr) : BOOLEAN;
```

The XTNDPutFile function presents a dialog box, similar to SFPutFile, with a pop-up menu of file formats. The user selects a format in which to save the current document. Like SFPutFile, XTNDPutFile repeatedly gets and handles events until the user names a file that can be written to or clicks Cancel. The function reports the user's choice via the Standard File reply record and through the SFParamBlock's ioResult.

Implementation of XTNDPutFile

As with GetFile, the first step in saving files is to set up your application's standards array. Since this does not differ in any way from the one used in GetFile, you may use the same array as long as your application writes the same file types that it reads.

After you have set up the standards array, you will set up the SFParamBlock. You set up all of the fields in the same manner as for importing, with the exception of those listed in Table 2-3.

Table 2-3 XTNDPutFile-specific parameters

In/Out (->/<-)	Parameter	Description
-->	applicNativeType	The application's native type (for use if the library cannot be found).
-->	prompt	A Pascal string that is written in the dialog box, above the pop-up menu.
-->	origName	A Pascal string used as the file's default name.
-->	buttonTitle	A Pascal string to replace the name of the Save button. If left NULL, "Save" is used.
<->	currentMenuItem	unused during a call to XTNDPutFile.
<->	currentSaveItem	The item chosen in the pop-up menu on a previous call to XTNDPutFile.
-->	showAllFiles	unused during a call to XTNDPutFile.

You should set applicNativeType to the application's native type.

Make `prompt` a pointer to any string which you want to put above the pop-up menu. This string generally prompts the user to select a filename and a file format. The string “Save As… ” is used if you set this pointer to nil.

Similarly, make `buttonTitle` a pointer to a string you wish to appear in the Save button. The name “Save” is used if you set this pointer to nil.

`origName` should be set to the default name of the file to be saved.

Set `currentSaveItem` to some reasonable value, usually 1 for the first call to `PutFile` (native format). After this, `currentSaveItem` can usually be ignored, since the call to `PutFile` sets `currentSaveItem` to the number of the last selected translator, and this is a good starting value for the next `PutFile` call. If you change the list of translators, you should probably set `currentSaveItem` back to 1, since the previous value may no longer be meaningful.

After you call `XTNDPutFile(&SFParams)`, check for any error or for a click in the Cancel box, and then proceed with writing out the file.

`showAllFiles` is not used in `XTNDPutFile`.

Using XTND to write the file

If the selected translator is one of the items in the `standards` array, then your application needs to use an internal translator to save the file using its own methods. Otherwise, use the XTND Translator to write the file.

Use a separate parameter block, `ExportParmBlock`, for exporting as defined in `ExportTranslator.h`. The first step in this procedure is to fill in the fields that are required for initialization of the export translator.

- *Note:* The buffer used for exporting is actually accessed via a *handle*, whereas the import translator uses a *pointer*.

You can assume that `exportPB` is an `ExportParmBlock` and `MinusOne` is a point. All other definitions are to the right of the associated field.

```
/* Initialize the export translator */
MinusOne.h = MinusOne.v = -1;
exportPB.result = &fserr;              /* short fserr; */
exportPB.refNum = &fnum;               /* short fnum; */
exportPB.textLength = &runlength;      /* long runlength; */
exportPB.textBuffer = NewHandle[256];  /* This is a HANDLE in export. */
exportPB.paraFmts = Paragraph;         /* Fixed Paragraph[9]; */
exportPB.tabs = tabs;                  /* TabSpec tabs[20]; */
exportPB.txtFace = &textface;          /* short textface */
```

```
exportPB.txtSize = &textsize;          /* short textsize; */
exportPB.txtFont = &textfont;          /* short textfont; */
exportPB.txtColor = &textcolor;        /* short textcolor; */
exportPB.txtJust = &textjust;          /* short textjust; */
exportPB.numCols = 1;
exportPB.topMargin = 0x480000;         /* 1 inch margin (72 points) */
exportPB.bottomMargin = 0x480000;      /* 1 inch margin (72 points) */
exportPB.left = 0x480000;              /* 1 inch margin (72 points) */
exportPB.right = 0x480000;             /* 1 inch margin (72 points) */
exportPB.gutter = 0xc0000;
exportPB.totalCharCount = (**te).teLength;
                                       /* total # of chars in doc */
exportPB.startPageNum = 1;
exportPB.startFootnoteNum = 1;
exportPB.rulerShowing = TRUE;
exportPB.doubleSided = FALSE;
exportPB.titlePage = FALSE;
exportPB.endnotes = FALSE;
exportPB.showInvisibles = TRUE;
exportPB.showPageGuides = TRUE;
exportPB.showPictures = TRUE;
exportPB.autoFootnotes = TRUE;
exportPB.footnotesExist = FALSE;
exportPB.pagePoint = MinusOne;
exportPB.datePoint = MinusOne;
exportPB.timePoint = MinusOne;
exportPB.smartQuotes = TRUE;
exportPB.fractCharWidths = TRUE;
exportPB.hRes = 72;
exportPB.vRes = 72;
exportPB.theReply = theReply;
exportPB.thisTranslator = *pChosenOne;
exportPB.currentStory = mainStory;
exportPB.printRecord = (THPrint)NewHandle(sizeof(TPrint));
                                       if (exportPB.printRecord)
{
      PrintDefault(exportPB.printRecord);
                                       /* This assumes the print     */
      PrValidate(exportPB.printRecord);   /* driver is already open  */
}
exportPB.headerStatus = exportPB.footerStatus = 0;            /*NONE */
```

The procedure for locating and locking down a translator for export is very similar to the one used during import. The only difference is in the name of the routine pointing to the export translator.

```
TransProcPtr     gExportTranslator;
```

```
TransDescrPtr    pChosenOne = exportPB.theChosenTranslator;
   if (fserr = XTNDLoadTranslator(pChosenOne, &gExportTranslator)) {
        AlertUser("\pUnable to load translator.");
        return;
   }

   /*   Now create the file so we can delete it.                    */
   /*   (Eliminates PMSP problem)                                   */
   Create(theReply.fileName, theReply.vRefNum, '????', '????');
   if ( FSDelete(theReply.fileName, theReply.vRefNum) )
   {
        AlertUser("\pUnable to delete file, probably write protected.");
        return;
   }

   Match = Trans.matches[0];
   fserr = Create(theReply.fileName, theReply.vRefNum, Match.docCreator,
            Match.docType);
   if (fserr)
   {
        AlertUser("\pUnable to create output file.");
        CloseResFile(resfile);
        return;
   }

   fserr = FSOpen(theReply.fileName, theReply.vRefNum, &fnum);
   if (fserr)
   {
        AlertUser("\pUnable to open output file.");
        CloseResFile(resfile);
        return;
   }
```

Now send the correct initialization sequence to the export translator. If you created a handle successfully for the print record, you must release the memory after the `exportCloseAll` directive.

```
exportPB.directive = exportInitAll;
(*gExportTranslator) (&exportPB);

if (exportPB.printRecord)
          DisposHandle( (Handle) exportPB.printRecord);
```

Now your application needs to parse the document and determine what segments you can export; within each of those segments, you need to determine the style runs.

The sample program shows the exporting of a styled TextEdit document. For this reason, the only concern is the main body of the document. Usually, you would cycle through the segments of the document. (The Font and Size menus are not enabled because they are used merely to display the current settings.) The sample program is set up for MPW C 3.2 and THINK C 4.0 and needs some modification if it is compiled in a different environment.

The sequence during export is important, so you can use a routine similar to the following one, which determines the next segment to be sent. This routine assumes that the `headerStatus` and `footerStatus` variables contain valid values. The routine continuously locates the next segment of the current document, and then attempts to call the translator with that segment. If the translator does not accept that kind of segment, it does not return an `exportAcknowledge` directive value. If the translator does not return an `exportAcknowledge` directive value, then cycle to the next portion of the document.

```
static void GetNextOpenDirective(thePtr)
register ExportParmBlock  *thePtr;
while (thePtr->directive != exportAcknowledge)
{
   done = FALSE;
   while (!done)
   {
        switch(thePtr->directive)
        {
             default:
                  done = TRUE;
                  break;

             case exportOpenRightHeader:
                  if (thePtr->headerStatus & rightPage)
                  {
             /*   Application-specific statements to point            */
             /*   to right header information            */
                  done = TRUE;
                  }
                  else
                  {
                       thePtr->currentStory = leftHeaderStory;
                       thePtr->directive = exportOpenLeftHeader;
                  }
                  break;

             case exportOpenLeftHeader:
                  if (thePtr->headerStatus & leftPage)
                  {
             /*   Application-specific statements to point            */
```

```
/*   to left header information                              */
          done = TRUE;
     }
     else
     {
          thePtr->currentStory = headerStory;
          thePtr->directive = exportOpenHeader;
     }
     break;

case exportOpenHeader:
     if (thePtr->headerStatus & everyPage)
     {
/*   Application-specific statements to point           */
/*   to header information            */
          done = TRUE;
     }
     else
     {
          thePtr->currentStory = rightFooterStory;
          thePtr->directive = exportOpenRightFooter;
     }
     break;

case exportOpenRightFooter:
     if (thePtr->footerStatus & rightPage)
     {
/*   Application-specific statements to point           */
/*   to right footer information           */
          done = TRUE;
     }
     else
     {
          thePtr->currentStory = leftFooterStory;
          thePtr->directive = exportOpenLeftFooter;
     }
     break;

case exportOpenLeftFooter:
     if (thePtr->footerStatus & leftPage)
     {
/*   Application-specific statements to point            */
/*   to left footer information             */
          done = TRUE;
     }
     else
```

```
            {
                thePtr->currentStory = footerStory;
                thePtr->directive = exportOpenFooter;
            }
            break;

        case exportOpenFooter:
            if (thePtr->footerStatus & everyPage)
            {
        /*  Application-specific statements to point            */
        /*  to footer information            */
                done = TRUE;
            }
            else
            {
                thePtr->currentStory = footnoteStory;
                thePtr->directive = exportOpenFootnote;
            }
            break;
        case exportOpenFootnote:
            if (GetNextFootnote(&thePtr->footnoteOffset))
            {
        /*  Application-specific statements to point            */
        /*  to footnote information           */
                thePtr->footnoteText = the_marker;
                done = TRUE;
            }
            else
            {
                thePtr->currentStory = mainStory;
                thePtr->directive = exportOpenMain;
            }
            break;
        case exportOpenMain:
            done = TRUE;
        /*  Application-specific statements to point            */
        /*  to main body information            */
            break;
    }
}
(*gExportTranslator)(thePtr);
switch(thePtr->directive)
{
    default:
        thePtr->directive = exportAcknowledge;
        break;
```

```
        case exportAcknowledge:
             break;
        case exportOpenRightHeader:
             thePtr->currentStory = leftHeaderStory;
             thePtr->directive = exportOpenLeftHeader;
             break;

        case exportOpenLeftHeader:
             thePtr->currentStory = headerStory;
             thePtr->directive = exportOpenHeader;
             break;

        case exportOpenHeader:
             thePtr->currentStory = rightFooterStory;
             thePtr->directive = exportOpenRightFooter;
             break;

        case exportOpenRightFooter:
             thePtr->currentStory = leftFooterStory;
             thePtr->directive = exportOpenLeftFooter;
             break;

        case exportOpenLeftFooter:
             thePtr->currentStory = footerStory;
             thePtr->directive = exportOpenFooter;
             break;

        case exportOpenFooter:
             thePtr->currentStory = footnoteStory;
             thePtr->directive = exportOpenFootnote;
             break;
        case exportOpenFootnote:
             thePtr->currentStory = footnoteStory;
             thePtr->directive = exportOpenFootnote;
             break;
   }
}
```

Once you get an `exportAcknowledge` directive value (by calling `GetNextOpenDirective`), you can proceed with writing that segment of the document via the translator. The `exportWriteText` directive is used to write the segment. As stated previously, you send information to the translator in "runs." These runs are either style runs or paragraphs, as in this example:

This is the first paragraph. Contained within is **one style** change.

The previous line ended with a return code, but no style change.

The two preceding lines above would be sent in four parts. The first would be the Garamond Plain text up to, and including, the space before the words “one style.” The next run would be just “one style,” with the associated text style changes. The third run would be from the space after “one style” up to and including the return code at the end of the line. The final run would be the entire second line.

This is how you set up the export parameter block. In this code segment, `textPtr` is the pointer to the text you wish to export. You need to copy it into a handle. You must then make the export directive `exportWriteText` and then you need to call the export translator repeatedly.

```
BlockMove (textPtr, *exportPB.textBuffer, *exportPB.textLength)
exportPB.directive = exportWriteText;
(*gExportTranslator) (&exportPB);
```

Note that you are not limited to a maximum of 256 characters in a single text run as you are during input. In practical terms, however, this is a reasonable maximum size for you to use.

After you have obtained the style run, you need to set up the paragraph information in `Paragraph`, the tab information in `tabs`, and the correct style information in the correct variables. Then you set up the handle that contains the text of the run, and call `exportWriteText`.

When you have sent all the text in the segment, close the appropriate segment and open the next one.

After you have performed an `exportCloseAll`, you must close the data fork of the document, open the resource fork of the document, and perform an `exportWriteResources` to complete saving the file. Following this call, you must close the resource fork of the document.

Finally, you just unlock and unload the translator, and close all the necessary files.

Matching a file to a translator

In some situations, you may need to match a file to a translator that can process it. You can use the function `XTNDMatchFile` contained in the XTND library to do this. You use `MatchTranslatorBlock` to provide the filename and other file information (volume reference number and directory ID) to the `XTNDMatchFile` procedure.The `XTNDMatchFile` procedure returns TRUE if it can find a translator to read the file that was passed to it; otherwise, it returns FALSE.

When you use this routine, there are three methods of specifying the translators to be matched against the file. In the first, the application sets the `allowFlags` field of the `MatchTranslatorBlock` to specify the type and usage of translators to be selected from the list of translators, which is maintained by the XTND library. Using the `standards` array, the application builds the list of translators to be matched against the file, much as it builds the pop-up menu in the `XTNDGetFile` call. This list is built only on the first call to `XTNDMatchFile` and when the `initFlag` field is set to TRUE; otherwise, the existing list is used. The `useTransList` flag must be set to FALSE when this method is used, and `chosenTranslator` and `theChosenTranslator` are ignored.

In the second method, the application supplies a list of translators. You do this by setting the `useTransList` field to TRUE, making `theChosenTranslator` a pointer to a `transDescribe` list, and setting `chosenTranslator` to the number of translators in the supplied list. The list of translators to be matched against the file is built from the supplied list and the `Standards` array . The list is built on the first call and when `initFlag` is set to TRUE; otherwise, the existing list of translators is used.

In the third method, the application specifies a single translator to be matched against the supplied file. You specify the translator by making `oneTrans` a pointer to the `TransDescribe` for the translator. When `oneTrans` is set, the existing list of translators is not used, the list is not rebuilt (even if `initList` is TRUE), and any other supplied parameters are ignored. The list, if built previously, is unchanged and is available on subsequent calls. This call is useful for determining which files in a group can be opened by a particular translator. It is designed to be very fast and simple.

When `XTNDMatchFile` returns TRUE, it also returns a value indicating which translator should be used to read the file. If this value is less than or equal to the number of standard types, then it is a “standard” file type (0 is “All available” on import and 1 through the value of `numStandard` are the standard types). Otherwise, its value represents an offset into an array of translators.

The `initFlag` field is used when repeated calls are to be made to `XTNDMatchFile` and the same list of translators is to be used for each. By setting this field to FALSE, you can use the existing list of translators and reduce the time required by this call.

In the following example, an application has set up the standard array as in MacWrite II and now wishes to determine if it can read a particular file. The `initFlag` field is set to TRUE. On subsequent calls, it would be set to FALSE, unless there was a reason to rebuild the list of translators. At present, you need to rebuilt this list only if the value of `allowFlags` changes or the standard translators change. Actually, the list is built on the first call, regardless of the value of `initFlag`.

```
MatchTranslatorBlock MFB;

MFB.allowFlags = *allowImport + *allowText;
```

```
MFB.numStandard = 3;
MFB.standard = Standards;
MFB.fileName = fileName;  /* Name of file to be opened */
MFB.vRefNum = vRefNo;     /* Volume of file to be matched */
MFB.ioDirID = fDirID;     /* Directory ID of file to be matched */
MFB.initFlag = TRUE;      /* Set this to FALSE on subsequent calls */

can_read = XTNDMatchFile(&MFB);
if (MFB.ioResult != 0) /*     Check for errors the library may return */
{
   /*   For example, we could display an alert here describing the error
or if we are a function, we could return that error.  XTNDMatchFile will
return an ioResult of noTransMatchErr if it cannot find a translator
that can read the file    */
   return (MFB.ioResult);
}
if (can_read)
{
if (MFB.chosenTranslator <= MFB.numStandard)
{
   /*   This is a “standard” file type.
   Set up for reading in the normal manner                              */
}
else
{
   /*
   This is a file which we can read via an XTND
   Translator MFB.theChosenTranslator points to the
   translator description     */

}
```

Selecting a list of translators

You use the function `XTNDSelectTranslators` to obtain, from the XTND library a list of translators that meet specified criteria. Several criteria for selecting translators, including version, language, type, and name, are available to the application, and you can apply them either individually or in combination. The application can use the list of translators (`transDescribe`) to create its own menus or to select a translator, or the list can be processed and used in an `XTNDGetFile` or `XTNDPutFile` call.

This function takes two parameters and returns an error status (0 if no error). The two parameters are, respectively, a pointer to `SelectParamBlock` and an empty handle. The `SelectParamBlock` specifies the selection criteria for the translators which are returned in the handle. The structure of the `SelectParamBlock` is specified in `XTNDInterface.h`. It has several fields that allow you to select translators.

The first two fields are `translatorVersion` and `translatorType`. The `translatorVersion` field specifies a version and language for the selected translators. See the section "XTND Localization," later in this chapter, for more information on identifying the language of applications and translators. To have all translators returned, set this field to 0xFF02. If a version is specified in the low byte of this field, only translators that match the version are returned. (Currently, the XTND system selects translators of only a single version, so a version *must* be specified.) If a language is specified in the high byte of this field, only translators that match the language are returned. To match the version and not the language, set the high byte of this field to 0xFF and put the desired version in the low byte.

Use `translatorType` to select a single type of translator resource, such as 'FLTI' or 'PFLT'. If this field is 0, it is ignored; otherwise, only translators whose type matches the type in this field are returned.

The next two fields are `includeFlags` and `excludeFlags`. These correspond to the `allowFlags` field and specify the type and use of translators that will appear in the `GetFile` and `PutFile` dialog boxes. To have translators of a particular type and use returned, set the bits of `includeFlags` that correspond to the type and use of the translator. For example, to have all text import translators returned, you would set `includeFlags` to `*allowText` + `*allowImport`. To have all translators of some type and usage returned, set this field to all 1's. To have all translators returned, set this field to 0.

u *Note:* If you set this field to all 1's, only those translators with some type and use setting are returned. If they have no type or use, they are not returned. Only those translators that match both the type and use are returned.

Use the `excludeFlags` field to reduce the set of translators that have been selected via `includeFlags`. A translator is rejected if it matches any of the types or uses specified in this field. For instance, you can exclude all text translators or all export translators (you can't exclude only text export translators, however). If you set this field to 0, nothing is excluded. If you set both `excludeFlags` and `includeFlags` to 0, then, all translators are returned (depending on the other selection fields, of course). For your use, the selection statement used to select translator types based on `translatorVersion`, `translatorType`, `includeFlags`, and `excludeFlags` is as follows:

```
curTypeFlags = transType.allowFlags & *allowTypeMask;
curUseFlags = transType.allowFlags & *allowUseMask;
```

```
includeTypeFlags = selectPtr->includeFlags & *allowTypeMask;
includeUseFlags = selectPtr->includeFlags & *allowUseMask;
excludeTypeFlags = selectPtr->excludeFlags & *allowTypeMask;
excludeUseFlags = selectPtr->excludeFlags & *allowUseMask;

if ((selectPtr->translatorType == 0
            || transType.Type == selectPtr->translatorType)
   && ((includeTypeFlags == 0) || (curTypeFlags & includeTypeFlags))
   && !(curTypeFlags & excludeTypeFlags)
   && ((includeUseFlags == 0) || (curUseFlags & includeUseFlags))
   && !(curUseFlags & excludeUseFlags))
{ /*    Accept translator type */ }
```

The variable `transType` is a record containing the type, version, and `allowFlags` setting for a particular type of translator resource such as 'FLTI' or 'FLGI', and `selectPtr` is a pointer to your `SelectParamBlock`.

The next two fields are `includeTrans` and `excludeTrans`, which are used just as `includeFlags` and `excludeFlags` are. These fields are compared to the `flags` field in the `TransDescribe` block for the translator. Thus, you use these fields to select individual translators, whereas you use the `includeFlags` and `excludeFlags` fields to select types of translators. Again, to include all translators regardless of whether they have any flags set, set `includeTrans` to 0. This is somewhat more useful in this case because translators need not have any of these flags set. If you set `excludeTrans` to 0, no translators are excluded. To include all translators regardless of their `flag` settings, set both of these fields to 0.

The next field, `transName`, is a pointer to a Pascal string (Str255). It specifies a name for the selected translator. Only translators whose names match the supplied name are returned. You should not select translators by name because the translator may be localized into another language, which might cause its name to change. Also, another translator with the same name might be substituted. To include all translators regardless of name, set this field to 0. The selection statement used to select translators based on these three fields is as follows:

```
version = selectPtr->translatorVersion & 0x00FF;
language = selectPtr->translatorVersion & 0xFF00;

if (((selectPtr->includeTrans == 0)
            || (checkTransPtr->flags & selectPtr->includeTrans))
   && !(checkTransPtr->flags & selectPtr->excludeTrans)
   && ((version == 0)
            || (version == (checkTransPtr->version & 0x00FF)))
   && ((language == 0xFF00)
            || (language == (checkTransPtr->version & 0xFF00)))
   && ((selectPtr->transName == 0)
   || !(IUCompString(selectPtr->transName, checkTransPtr->name))))
{ /*    Accept translator */ }
```

The variable `checkTransPtr` is a pointer to the `TransDescribe` block for the translator being considered.

The next field is `PBVersion`, which should be set to `one` for this version of the `SelectParamBlock`.

The final field is `menuSortFlag`. This is used to indicate whether the list of translators being returned should be sorted and have duplicates removed in the same manner as the XTND library does when creating a list for the `GetFile` or `PutFile` dialog box. If this field is set to TRUE, the list is sorted and has duplicates removed. If this field is set to FALSE, the list contains all translators that met the selection criteria in the order they were found. If you plan to use this list in a menu and don't wish to sort it yourself, you should set this field to TRUE.

Rebuilding the list of available translators

The function `XTNDRebuildTransList` rebuilds the list of installed translators while the application is running. This is useful if the user needs to install or remove translators while using an application. The alternative to this call is to quit the application, discard the `XTND Translator List` file, and restart the application. This routine is also useful in ensuring that the list of translators is current, if there is any doubt.

XTND localization

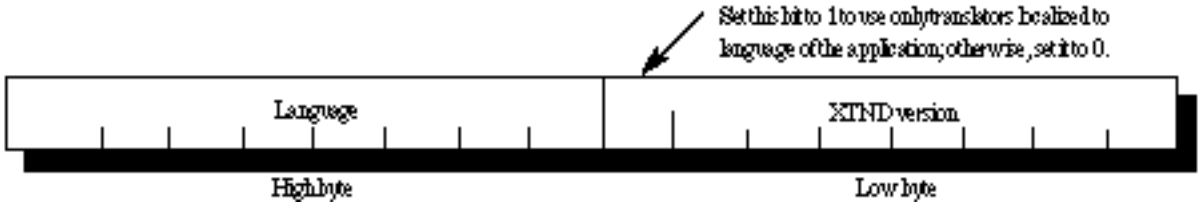
Translators, like applications, can potentially display language-specific user interface elements, such as dialog boxes. For this reason, future translators may need to be localized into other languages. Because translators are installed separately from applications and are supplied by various vendors, the user's machine may contain translators and applications in a multitude of languages. It is even possible, for example, that two translators (one in English, the other in French) for the same file format might exist on a single machine.

The XTND 1.3 release introduces a set of conventions allowing applications and translators to identify their language to the XTND System. The XTND System has been enhanced so that it automatically matches the application's language to the translator language and, when the choice of language is ambiguous, appends language names (in parentheses) to the translator names displayed in the pop-up menu list.

Application localization

You can identify the language of an application to the XTND System by passing that language (obtained from the 'vers' resource) in the high byte of the version when calling `XTNDInitTranslators`. You may pass 0xFF in this byte if you wish to associate the application with no language. Set the high bit of the low byte of the version to 1 if you wish to use only translators that are localized to the language of the application. This leaves only seven bits for the XTND version, which should be sufficient. (The current version is 2.) If you don't wish to be bothered with this, pass only the XTND version, as was done in previous XTND releases. XTND interprets this as an English application that can use translators localized into other languages. This should present no problems for existing applications. See Figure 2-1

Figure 2-1 XTND version (short integer)



Translator localization

Translators need not do anything different unless they have been localized into another language. You can identify the language of a translator to the XTND System by changing the version field in the 'FTYP' resource. The high byte of this field contains the language of the translator. The low byte contains the version of the translator. Set the high bit of the low byte of the version to 1 if the translator can be used only with applications in the same language as the translator. Additionally, the flag `ftypIsLocalized` identifies the translator as being localized. If the translator has been localized, this flag should be set. This flag allows the Claris XTND System to determine whether a translator with 0 in the high byte of the version is an English translator or one that has not been localized.

Language matching algorithm

Here is a pseudo-code description of the algorithm the XTND System uses to process and display the translator list.

```
If the application-language is 0xFF:
   List all translators in all languages.
   Append the 'language' string to each translator name when presenting
      choice to the user
If there is only one language version translator for a given format:
   Use it automatically
   If it does not match the language of the application, and it does not
      match the language of the system, append the 'language' string
      when presenting to the user, otherwise don't append it.
If more than one language-version of a given translator is available:
   If one of them matches the application-language:
      Use it automatically; don't present the 'language' string to the
            user
   Else if one of them matches the system-language:
      Use it automatically; don't present 'language' string to the user
   Else :
      Present all the available language-versions of a given translator
      Append the 'language' string to each translator name when
            presenting choice to the user.
```

- *Note:* Languages are matched according to an internal list of equivalent languages. Thus, Canadian French, French French, and Swiss French are equivalent. The translator that best matches the application or the system language is selected.

- *Note:* If a translator is not localized, a *language* string is not displayed.

- *Note:* If two translators with the same name are encountered in the search process, and one is localized and the other is not, the first one found is displayed.

Chapter 3 Writing XTND Translators

Chapter 3 describes the process of using the XTND System from the translator's point of view. First, the structure of a translator file is described. Next, the simplest kind of translator, for PICT import, is described. After that, the protocol for importing text is described in detail. Finally, the protocol for exporting text is described.

Translator file structure

A translator file may contain several resources. The first type is 'FTYP'. It contains all the information that the XTND System needs to determine whether a translator can read a document.

The second resource is a code resource. This code resource contains the code that actually performs the translation of the foreign file format. Its resource type and identification (ID) are determined by the 'FTYP' resource. Example resource types are 'FLTI' (text import translator), 'FLTE' (export translator), and 'PFLT' (picture import translator).

The translator file may include a number of optional resources that are generally not required. An 'FDIF' resource is a code resource invoked by the XTND System to determine whether the document currently selected is readable by this particular translator. Another optional resource is 'FINI', which is discussed later in this chapter.

The 'FTYP' contains the resource type and ID of the translator code (the 'FLTI', 'FLTE', or 'PFLT') resource and the ID of the 'FDIF' resource. The ID number of the 'FINI' resources must be 0.

The 'FTYP' resource

The XTND System uses the translator's 'FTYP' resource to establish the mapping from document versions and file types to a specific translator code resource. Files that share the same creator and type, but differ in the internal format, can be mapped into separate translators. Conversely, files that have a distinct creator and/or type, but share the same file format, can be mapped into a single translator. A template to be used for creating 'FTYP' resources with ResEdit™ is included with this kit.

For example, all Microsoft® Word documents have the same creator and type, but their internal document formats differ. You could build one translator to read all these different formats, but it would have to be a very large translator. Instead, you can use the fact that Microsoft has changed its internal document version number. This allows you to create separate translators for Word 3 and Word 4 files; the 'FTYP' resources in these translators determine whether they can be used to open a particular file.

As another example, both MacDraw® II documents and MacDraw II stationery have the same file format, yet different file types. Instead of writing two different 'FTYP' resources (resulting in two pop-up menu entries), you can put multiple matches into a single 'FTYP' resource.

u *Note:* Every 'FTYP' resource must have at least one `match`.

The characteristics of the translator are identified by the flags field in the 'FTYP' resource. Six flag bits are currently defined:

- `ftypIsSpecial` designates a translator as a special translator. Usually, translators determine which files should appear in the open dialog box, and the XTND System checks all translators to see if one of them can open a selected file. A translator may be unable to determine precisely which files it is able to open. This problem typically occurs with files from PCs and other computer systems that do not have their file type and creator set to meaningful values when they are moved to the Macintosh. If the version bytes and 'FDIF' resource cannot discriminate between files that the translator can read and those that it cannot, the translator will attempt to open files it cannot read.

  If the wrong translator attempts to open a file, the result is considerable confusion for the user. When the `ftypIsSpecial` flag is set, the translator is considered a special translator. When the user selects a file to be imported, all of the non-special translators are tried first, then the standard (application native) types, and then the special translators.

  The drawback to designating translators as special is that opening files with these translators is slightly less automatic. Since files to be opened by such translators usually show up as “text” on the Macintosh, they are opened as text if the application reads text files (most do), after which the user may select a more appropriate translator and open them again.

- `ftypNeedsResources` specifies that the translator cannot be called unless its resource fork is open because it needs to read resources out of its resource fork (this is supported by the field in the `TransDescribe` record, `resRefNum`, which contains the refnum of the resource fork of the translator, or 0 if the resource fork is not open).

- `ftypIsLocalized` specifies that the translator has been localized to a particular language.

Translators must be able to access their resources when they are called. The application should ensure that the resource fork of the translator is open when the translator is called, if this is possible. Translators, by contrast, should not depend on being able to access their resources. With the exception of translators that have their `ftypNeedsResources` bit turned on (those that have additional code resources, for example), translators should be able to execute successfully even if they fail to read or write their resources. You can ensure this in the case of preferences, for example, by substituting default values for the stored preferences.

- *Note:* Some existing translators (notably those from DataViz™) will not work unless they can read from their resources, and they do not change the current resource file when they are called. In order to use these translators, the application must not only ensure that the resource fork of the translator is open, but also that the translator is in the list of currently searched resource files.

Translators can access their resources by setting themselves to be the current resource file using the `resRefNum` field in the `thisTranslator` field of the import or export parameter block. This field is set before the translator is called, and if it is non-zero then the resource fork is open. Translators should not write to their resources; they should store preferences in a preferences file stored in the Preferences folder. All translators from a single vendor can use a single preference file by having each translator store its preferences in a unique resource within the preference file. Before returning, the translator should restore the current resource file to the value it had when the translator was called.

'FINI' and 'FDIF' resources are not intended to read or write to their resources when they are called, and there is no way for them to tell if their resource fork is open or if it is the current resource file.

Translators can be localized into other languages. Most existing XTND translators do not have a user interface of their own, but future translators may require a preferences dialog box, or perform a language-specific translation. Translators can identify their language to the XTND System by changing the version field in the 'FTYP' resource. Please refer to the section “XTND Localization” in Chapter 2 for details about localization of translators.

The 'FTYP' resource format is shown in Appendix A.

Primary version check method—Version bytes

The version bytes in an 'FTYP' resource are used to distinguish among files having different versions but the same creator and file type attributes. This method of version checking assumes that the document has an internal version number at some fixed offset from the beginning of the file.

You can check up to 16 version bytes, where the number of bytes to check is recorded in `numVersBytes`. The offset is measured from the beginning of the file and is stored in `versBytesOffset`. The bytes required for a match are stored in `versBytes`. The translator cannot read the file unless the bytes stored here match those at the given offset in the file.

The MacWrite 5.0 file translator, for example, cannot read the file unless the first long word in the document has the value 6. The 'FTYP' resource for MacWrite 5.0 Import, numVersBytes is set to 2, versBytesOffset is set to 0, and the versBytes array shows that the first two bytes should be 00 06.

Secondary version check method—The 'FDIF' resource

If the version bytes method cannot be used or the values returned are incomplete, then an alternative is to use the 'FDIF' method instead (or in addition).

An 'FDIF' code resource simply returns a value (usually 1 or 2), according to whether a particular document is of the correct type. It can do whatever is needed in order to perform this check. The application opens the data fork of the file with read-only access and passes two parameters to the 'FDIF' code resource.

Here is the prototype of this function:

```
void main(short *TwoWayInfo, ResType Creator, TransDescribe
            *Translator);
```

On entry, `*TwoWayInfo` is the file reference number of the open document, `Creator` is the creator of the file, and `Translator` is a description of the translator being called. On exit, the value of `*TwoWayInfo` is 2 if the document is of the correct type but is the wrong version. If the value of `*TwoWayInfo` is 1, then this is the correct version of this document, and this translator can read the file. If the value is anything else, then the application will go on to the next translator.

See Appendix D for a simple example from the MacWrite 5.0 import translator.

An alternative—The 'FINI' resource

A file translator can also create 'FTYP' resources dynamically at startup. This is accomplished through a 'FINI' resource. This mechanism is useful if a file translator needs to retrieve information from another source (for example, another file) to determine which 'FTYP' resources it can support. In this case, the 'FINI' takes the information from the file and generates the appropriate 'FTYP' resources.

A 'FINI' code resource returns a count and a handle to an array of `transDescribe`. (See `Translator.h`.) The `transDescribe` structure is almost identical to the 'FTYP' structure, and 'FTYP' resources are changed into `transDescribe` structures when they are read into memory. The resource ID of the 'FINI' resource must be 0.

Here is the prototype of a 'FINI' resource:

```
void main(TransDescribe **FTYPHandle, short *FTYPCount);
```

On entry, `FTYPHandle` is a handle of size 0 and the value of `*FTYPCount` is 0. On exit, `FTYPHandle` should contain an array of `transDescribe` structures, and `*FTYPCount` should denote how many `transDescribe` structures are contained by `FTYPHandle`.

Translator programming interface

Translators are written as code resources. All resources related to the translator must be placed in a file. The translator file must be of type 'Fltr'. Remember, case is important. When built, the translator file must be placed where the XTND System can find it (see the section “Locating the XTND library” in Chapter 2 for more information on this subject).

The translator interface requires the main routine, which is called by the application. This function handles all the translator commands, from initialization to disposal of any memory that was used. The translator’s main routine is passed a pointer to a parameter block much as a Macintosh `PBRead` call is. This parameter block contains a directive value that tells the translator what action to take.

Errors

A translator returns error codes via the `result` field. If no error occurs, the translator should not change this field. In general, the translator should be able to handle any unknown directive it receives (by doing nothing) without returning an error. The translator should not return an `importAcknowledge` directive in response to an unknown directive, nor should it return an error.

If a file error occurs while the file is being retrieved from disk, the file system error should be returned to the application. Similarly, if a memory error or other system error is received, it should be returned. In both cases, the application should treat the error as if it had occurred within the application itself. The import (or export) should be terminated unsuccessfully, and the application can present an alert box describing the problem.

If the translator cannot import the file because it does not understand the file format, the translator should return the error code `badImportFileErr`, and the application should display an alert box stating that the translator was unable to read the file. The only exception to this is if the translator handles the error itself and puts up its own dialog box or takes another action to inform the user that the import could not be completed. In this case, the error code `translationCanceledErr` is returned, and the application should stop the import and take no other action. At present, these codes are returned in the error field and might be confused with system errors. Neither `badImportFileErr` nor `translationCanceledErr` conflicts with any system error codes, but this may not always be true.

Writing a PICT (or Picture) import translator

The PICT import translator is probably the simplest kind of translator that XTND supports. In general, the application opens both the data and resource forks of the document and then calls the translator and requests a picture. The translator then returns a handle to the requested picture.

When the application calls the translator, it passes a pointer to a picture translator parameter block. This parameter block contains fields in which the application passes the reference numbers of the data and resource forks of the file to be read. The translator uses other fields in the parameter block to return any error that may have occurred and, if all goes well, the graphic picture the user wanted.

The application may request either a QuickDraw picture or a PostScript® picture from the translator. If the translator has the requested picture representation, it returns it, otherwise, it returns a nil handle.

The application is responsible for closing the data and resource forks. The translator creates the picture handle, and the application disposes of it.

See Appendix B for a description of the picture translator parameter block and Appendix D for the the code for reading a PICT file.

Writing a text import translator

A text import translator reads a specified document and provides the application with information that describes the contents of that document. Because the application expects the information in a certain order, a number of typical document segments have been defined. The application requests these segments (header, footer, main body) in a fixed sequence.

Writing the translator

The text import interface requires a `main` routine, which is called by the application. This routine handles all the translator commands, from initialization to disposal of any memory that was used. The translator's `main` routine is passed a pointer to a parameter block, much as the Macintosh `PBRead` call is. Within this parameter block is a directive value that tells the translator what action to take.

See Appendix B for the definition of the import translator parameter block, Appendix C for the directives that the application can pass (through `directive`) during import, and Appendix D for a sample entry point for a text import translator (taken from the MacWrite 5.0 import translator).

Import translator actions

This section describes the actions the application expects the translator to perform when it passes specific selection information. The application can call your translator from one of two places: from the Open or from the Insert File menu item. If from the former, the translator sets up the page information, width of columns, number of columns, headers, footers, and starting page (and footnote) number. However, if the translator is called from Insert File, only the text and graphics are accepted, and the rest is discarded. There is no way of knowing which call invoked the translator, and it is expected to perform the same in both situations. If the call was invoked by Insert File, the application may not call any header or footer routines. Two fields in the parameter block control which portion of the document is currently being imported. They are the `directive` and `currentStory` fields. The `directive` field informs the translator what task is to be performed, and the `currentStory` field describes which portion of the document is being imported.

directive Values

- `importGetResources`

  If your document has a resource fork, then your translator will receive an `importGetResources` call initially; otherwise, the first call it receives is `importInitAll`. Your translator should be able to deal with an `importGetResources` call (by doing nothing), even though it is not expecting the document to have a resource fork.

  At this point in the sequence, the application has opened the document's resource fork and is calling the translator to read in any necessary resources from the document. The translator should read all the required resources and release them, since the application closes the resource fork when this call is completed. If the translator does not need any resources, then it should do nothing.

- `importInitAll`

  At this point, you should initialize your global variables. This directive is always given, whereas `importGetResources` is given only if the document has a resource fork. The variables that you should initialize within this call are as follows:

  ```
  importParms->autoHyphenate
  importParms->topMargin
  importParms->bottomMargin
  importParms->leftMargin
  importParms->rightMargin
  importParms->gutter
  importParms->startPageNum
  importParms->startFootnoteNum
  importParms->rulerShowing
  importParms->doubleSided
  importParms->titlePage
  importParms->endnotes
  importParms->showInvisibles
  importParms->showPageGuides
  importParms->showPictures
  importParms->autoFootnotes
  importParms->printRecord
  importParms->numCols
  ```

The `printRecord` variable is a special case. If you use a print record, you have to create the print handle in your initialization routine and pass back that handle. You do not dispose of it because that is done for you. If you do not use a print record, leave this entry NULL.

The general idea behind the remainder of the import process is that you initialize a segment of the document and then fill that segment with information. You do this by using `directive` and `currentStory`. The application will loop through the segments, initializing each segment, getting text, and then closing the segment.

- `importInitRightHeader`
- `importInitLeftHeader`
- `importInitHeader`
- `importInitRightFooter`
- `importInitLeftFooter`
- `importInitFooter`
- `importInitMain`
- `importInitFootnote`

The application calls the translator; the `currentStory` and `directive` fields indicate where to begin to import information. The translator begins checking for a right header by passing `importInitRightHeader` in `directive`, and `rightHeaderStory` in `currentStory`. If the document that is being translated contains a right header, pass back `importAcknowledge` in `directive`. The application continues through the above values until it receives an `importAcknowledge` from the translator.

The application cycles through the stories in the order listed before the preceding paragraph. Each story is requested only once, with the exception of `importInitFootnote`, which is called for each footnote passed during the import of the main body.

u *Note:* The application may continue to request footnotes until the translator indicates that no more are available. It indicates this by not returning `importAcknowledge` when it receives an `importInitFootnote` directive from the application.

u *Note:* XTND presently allows footnote references to appear only in the main body. The translator should not return any footnote reference characters in any other stories.

To reiterate, after the main body has been imported (that is, after `importCloseMain` is passed), the application calls your translator once for each footnote in the following cycle: `importInitFootnote`, `importGetText`, and `importCloseFootnote`.

- `importGetText`

  When `directive` is set to `importGetText`, the translator is expected to return the contents of the `currentStory`. Up to 256 bytes of text is returned at a time. Each "style run" should be returned as a separate object. A style run is any consecutive part of a story that has the same paragraph and character attributes.

  For a regular style run, the following fields should be returned:

`importParms->textLength`	The number of characters in the selection.
`importParms->textBuffer`	The characters in the selection.
`importParms->txtFace`	The face of the selection.
`importParms->txtSize`	The font size of the selection.
`importParms->txtFont`	The font family ID of the selection.
`importParms->txtColor`	The XTND 1.3 color of the selection.
`importParms->txtJust`	The XTND 1.3 justification of this paragraph.
`importParms->paraFmts`	A pointer to the paragraph format for this paragraph.
`importParms->tabs`	A pointer to the tab information for this paragraph.

If the translator is passing any of the special characters in italics (see the section "Data Types and Conventions" in Chapter 1), such as page number, footnote reference, or picture, then `textLength` must be equal to 1 and only one character is returned via `textBuffer`. Information specific to each of these special characters may also be passed to the application via the `miscData` long word. (See "Special characters," later in this chapter, for more details on how to pass these characters.)

After returning all of the information in each story, the translator should return `importAcknowledge` in response to the next `importGetText` call from the application. No text should be returned, and the `textLength` field should be set to 0 when `importAcknowledge` is returned. This response tells the application that the translator has completed this story, and that the application can request the next one. Prior to moving to that story, the application sends a "close" story command to the translator, as follows:

- `importCloseRightHeader`
- `importCloseLeftHeader`
- `importCloseHeader`
- `importCloseRightFooter`
- `importCloseLeftFooter`
- `importCloseFooter`
- `importCloseMain`
- `importCloseFootnote`

These calls allow your translator to dispose of any memory it used while importing the story. These calls are separate because the application may run out of memory or disk space during an import, and you may wish to close the translator in a different manner than usual.

- `importCloseAll`

A call to `importCloseAll` signifies that the import has been completed. At this time, the translator must clean up any handles or pointers it has created and dispose of any resources that it read in earlier. This is always the last call made to the translator during an import. The application makes this call even if the translator has returned an error to a previous call.

Special characters

If the special character in `textBuffer` is `floatingPict` (0x04), then `miscData` is a handle to `PictMisc`, which is defined in Table 3-1.

Table 3-1 `PictMisc` definition

Length	Name	Description
4 bytes	`thePicture`	A QuickDraw picture handle.
4 bytes	`pictSize`	The size of the picture.
8 bytes	`destRect`	The user-supplied picture display rectangle that the picture is scaled to after it is cropped to the OrigRect.
8 bytes	`origRect`	The picture display rectangle that the picture is cropped to.
38 bytes	`reserved`	Fill with 0's.

Both this 62-byte structure and the picture handle must be created by the translator.

Usually, for all other special characters, `miscData` is a long word containing 0—indicating the default for that special character. However, Table 3-2 details other possible values of `miscData`.

Table 3-2 `miscData` values

Value		Special character	miscData Value
2	(0x02)	Page number	If non-zero, this is a “current of total” page number character. If 0, this is a simple page number character.
4	(0x04)	Picture	PictMisc handle (see previous section).
5	(0x05)	Footnote reference	Zero (see following section).
21	(0x15)	Short date	If nonzero, this date character is never updated, and this field reflects the time at which this character was inserted (in seconds since January 1, 1904). If 0, this character always reflects the current date.
22	(0x16)	Abbreviated date	Same as Short date.
23	(0x17)	Long date	Same as Short date.
24	(0x18)	Abbreviated with day date	Same as Short date.
25	(0x19)	Long with day date	Same as Short date.
26	(0x1A)	Time	If nonzero, this time character is never updated, and this field reflects the time at which this character was inserted (in seconds since January 1, 1904). If 0, this character always reflects the current time.

“Offset” special characters

The document being imported may have date, time, or page characters that are offset into the header or footer by some QuickDraw point. If so, the translator must return the offset (as a point) from the top-left corner of the story in the respective parameter block field (that is, `datePoint`, `timePoint`, or `pagePoint`). When the application reads in the story, it inserts the correct character into the story at the specified point.

Footnotes

XTND allows footnotes only within the main story. The application should handle the automatic numbering of the footnotes that are sent by the translator if the `autoFootnotes` field is set to TRUE. If `autoFootnotes` is set to FALSE, then the text string with which the footnote reference is to be marked must be returned by the translator in the `footnoteText` field. The `footnoteText` field is a Pascal-based string containing no more than nine characters.

Remember that the text of the footnote is not inserted at this time but after the main body has been completed. After the main story has been read in, the application calls the translator once for each footnote. At that time, the application invokes the translator with a call to `importInitFootnote` and then a call to `importGetText`. Remember that each footnote must be returned in the order in which it appears in the main body.

Mapping objects and attributes

Because the current XTND document model provides only one header and footer, multiple headers and footers, found in some file formats, are ignored. Your translator also has to map your file format's text attributes to similar XTND attributes. For example, the original application may support double underlining but use a different value than XTND does.

Writing a text export translator

To understand this section, you should first understand the previous section, "Writing a Text Import Translator," because a number of the concepts are the same.

A text export translator writes to a specified document, which the application creates and opens and for which it supplies the translator with a `refNum`. The application then provides information that describes the contents of that document. The application supplies the information in a prescribed sequence, and the translator must be able to accept it in that sequence. The translator is repeatedly called with information regarding the document, and the translator builds a file from the information provided.

Writing to the translator

The text export interface requires a `main` routine, which is called by the application. This function handles all the translator commands, from initialization to disposal of any memory that was used. The translator's `main` routine is passed a pointer to a parameter block, much as a Macintosh `PBWrite` call is. This pointer contains a directive that tells the translator to supply a particular segment of information.

See Appendix B for the definition of the export parameter block, Appendix C for the directives that the application can pass (through `directive`) during export, and Appendix D for a sample entry point for a text export translator (taken from the header of the MacWrite 5.0 export translator).

Export translator actions

This section describes the actions the application expects you to perform when your translator is passed specific selection information. Two fields in the parameter block control which portion of the document is currently being exported. They are the `directive` and `currentStory` fields. `directive` informs the translator which task is to be performed, and `currentStory` describes which portion of the document is being exported.

`directive` values

- `exportInitAll`

This is the first call received by your export translator. It allows you to set up your translator global variables and other variables. The application sets up the following elements of the parameter block, thus you can determine the layout of the document you are creating:

```
exportParms->result
exportParms->refNum
exportParms->topMargin
exportParms->bottomMargin
exportParms->leftMargin
exportParms->rightMargin
exportParms->numCols
exportParms->gutter
exportParms->startPageNum
exportParms->startFootnoteNum
exportParms->rulerShowing
exportParms->doubleSided
exportParms->titlePage
exportParms->endnotes
exportParms->showInvisibles
exportParms->showPageGuides
exportParms->showPictures
exportParms->autoFootnotes
exportParms->printRecord
exportParms->headerStatus
exportParms->footerStatus
exportParms->totalCharCount
exportParms->autoFootnotes
```

`refNum` contains the reference number of the data fork of the file to which you are writing.

The `headerStatus` and `footerStatus` variables are status bytes that tell whether headers and footers exist, in case any special preparation must be done for documents containing headers and footers. The associated information is shown in Table 3-3.

- **Table 3-3** `headerStatus` and `footerStatus` variables

Bits	Meaning
0	TRUE if every page.
1	TRUE if left page.
2	TRUE if right page.

- `exportOpenRightHeader`
- `exportOpenLeftHeader`
- `exportOpenHeader`
- `exportOpenRightFooter`
- `exportOpenLeftFooter`
- `exportOpenFooter`
- `exportOpenFootnote`
- `exportOpenMain`

These commands are called in the exact sequence listed here to initialize each segment of the document. The application is searching for the first valid document portion that the translator can process. If your document can support a particular segment (for example, right headers), the translator should return `exportAcknowledge` in the `directive` field when it receives this Open call. As with import, the translator should take no action when it receives an unknown directive and should not return an error; nor should it return `exportAcknowledge`.

When the translator returns an `exportAcknowledge`, the application responds with an `exportWriteText` command, which is described in the following section. At the end of the story, the application sends the relevant `exportClose` command and cycles through to the next story type. As with the import translator, footnotes are the exception, in that footnotes are sent until they have all been exported.

- `exportWriteText`

A style run is any consecutive part of a story that has the same paragraph and character attributes. The application fills in the following fields in the parameter block:

`exportParms->textLength`	Number of characters in the selection.
`exportParms->textBuffer`	A handle containing the text.
`exportParms->txtFace`	Pointer to the face of this selection.
`exportParms->txtSize`	Pointer to the font size of the selection. (This is actually the QuickDraw font size multiplied by four.)
`exportParms->txtFont`	Pointer to the font family ID of the selection.
`exportParms->txtColor`	Pointer to the XTND 1.3 color ID of the selection.
`exportParms->txtJust`	Pointer to the justification of this selection.
`exportParms->paraFmts`	Pointer to the paragraph information (see “Writing a graphic import translator,” earlier in this chapter).

`exportParms->tabs`	Pointer to the tab information (see Import).
`exportParms->pagePoint`	QuickDraw Point location of the page number character.
`exportParms->datePoint`	QuickDraw Point location of the date number character.
`exportParms->timePoint`	QuickDraw Point location of the time number character.

Special characters

Special characters are dealt with differently during export than they are during import. During import, all special characters are passed individually, whereas during export, page, date, and time characters can be passed in the text stream. XTND does not currently allow the export of the extended data associated with these special characters.

If a picture is being exported, then the Picture character is passed by itself, and the field `picture` contains a `PicHandle`, and the field `pictRect` contains a rectangle that is the duplicate of the QuickDraw subfield `picFrame`.

If the main story is currently being exported and a footnote character is encountered, then the footnote character is also sent individually. The additional field used by the footnote is `footnoteText`, which is a Pascal string. If this string is empty (the length byte is 0), then the footnote is automatically numbered; otherwise, this string is the footnote marker.

Footnotes

Footnotes are treated as special cases during export, just as they are during import. During import, footnotes are passed after the main body, but during export, footnotes are passed before the main body. This is done for files that must place the footnotes within the main body text, since such translators must store the text of either the main body or the footnotes.

Each footnote is a story by itself and is passed as such. Each time the translator is called with `directive` set to `exportWriteText`, the `currentStory` is `footnoteStory`, `footnoteOffset` is the offset into the document where this footnote occurs. The application sets up both `footnoteOffset` and `footnoteText` when it invokes the translator with a call to `exportOpenFootnote`.

Closing

Upon completion of each story, the application calls the translator with the appropriate `Close` directive. In order, they are as follows:

- `exportCloseRightHeader`
- `exportCloseLeftHeader`
- `exportCloseHeader`
- `exportCloseRightFooter`
- `exportCloseLeftFooter`
- `exportCloseFooter`
- `exportCloseFootnote`
- `exportCloseMain`

  These calls signify the end of that story, so the translator can close the story in your document.

- `exportCloseAll`

  This call is sent after all the main body has been read and closed. At this time, the translator must dispose of any memory that was used. This is the last chance to write to the data fork of your document. The application should make this call even if the translator has returned an error to a previous call.

- `exportWriteResources`

  Finally, the translator is called with this directive. If you must write any resources to your document, this is the time to do it. When this call is sent, `refNum` is a pointer to the file reference number of the resource fork of your document. The translator must not close the resource fork of the document; the application will close the resource fork.

Appendix A 'FTYP' resource format

n **Table A-1** 'FTYP' resource format

Offset		Length (bytes)	Description	
0	(0x0000)	2	`version`	Translator revision number. Currently 2 (0x02).
2	(0x0002)	4	`translatorType`	Translator type. Examples are: 'FLTI' = Text Import Translator. 'FLTE' = Text Export Translator. 'PFLT' = Picture Import Translator.
6	(0x0006)	2	`codeResID`	The resource number of the translator type.
8	(0x0008)	2	`FDIFResID`	'FDIF'' resource number. If this is set to -2, this translator is an export translator (and as such needs no 'FDIF'' resource). If set to -1, there is no 'FDIF'' resource.
10	(0x000A)	2	`numVersBytes`	Number of version difference bytes.
12	(0x000C)	4	`versBytesOffset`	Offset into document of version difference bytes.
16	(0x0010)	16	`versBytes`	Byte values for this translator.
32	(0x0020)	2	`appWDRefNum`	`WDRefNum` of the application folder. This field is set internally by the XTND System.
34	(0x0022)	2	`unused1`	Reserved for future use—must be 0 for this version.
36	(0x0024)	2	`pathLength`	This field is set internally by the XTND System.
38	(0x0026)	2	`flags`	This field contains bits that are used to designate the characteristics of this translator. Flag bits are defined as follows: Value Meaning 0x0001 `ftypIsSpecial`. Designates this as a special translator.

(Continued)

n **Table A-1** 'FTYP' resource format (Continued)

Offset		Length (bytes)	Description	
				0x0002 ftypHasPreferences. Indicates this translator has a preferences dialog box. 0x0004 ftypNeedsResources. Means this translator cannot run without having its resources available. 0x0008 ftypWritesResources. Indicates this translator will try to write to its resources. 0x0010 ftypOnlyPreferences. Indicates this can be used only to set its preferences. 0x0020 ftypIsLocalized. Means this translator has been localized into a particular language.
40	(0x0028)	2	transIndex	Index of this translator in the list stored by the 'FINI' resource. This field is set internally by the XTND System.
42	(0x002A)	2	resRefNum	Reference number of the resource fork of this translator. This field is set internally by the XTND System.
44	(0x002C)	4	directoryID	Directory ID of the folder containing this translator. This field is set internally by the XTND System.
48	(0x0030)	2	vRefNum	Volume reference number of the volume containing this translator. This field is set internally by the XTND System.
50	(0x0032)	32	fileName	Name of the file containing this translator. This field is set internally by the XTND System.
82	(0x0052)	2	numMatches	Number of matches (*n*, described below).

(Continued)

n **Table A-1** 'FTYP' resource format (Continued)

Offset	Length (bytes)	Description	
84 (0x0054)	10**n*	`matches`	A `match` has the following format: Length — Meaning 4 — Document creator. For example: 'MACA' = MacWrite 5.0 4 — Document type. For example: 'WORD' = MacWrite 5.0 1 — (Boolean) Set to TRUE if the creator must match as well as the file type. 1 — This bit field allows particular bytes of the file type (bits 4-7) and creator type (bits 0-3) to be ignored. If a bit is set, the corresponding byte is ignored. For example, a translator that accepted all files of type 'XAAA' and creator 'CCXC', where 'X' doesn't matter, would set bits 7 and 1 of this byte.
84 + 10*n	32	`name`	Name of this translator as a Pascal string. This name appears in pop-up menus displayed by the XTND System.

Appendix B **Parameter block formats**

Picture translator parameter block

Table B-1 depicts the picture translator parameter block.

Table B-1 Picture translator parameter block

	Offset	Length (bytes)	Description	
0	(0x0000)	2	`result`	On entry, it is set to `ioErr`. Remember to set the field to `noErr` if you were successful.
2	(0x0002)	2	`dataRefNum`	File reference number of the data fork of the graphics file.
4	(0x0004)	2	`resRefNum`	File reference number of the resource fork of the graphics file.
6	(0x0006)	4	`thePicture`	On entry, not defined; on exit, it should contain a valid QuickDraw picture handle.
10	(0x000A)	2	`directive`	The desired action to be taken by the translator. MacWrite II calls a 'PFLT' translator only once and sets this field to `pictGetPicture (1)`. Other Claris applications may set this field to other values. Your 'PFLT' should respond only to directives that it understands.

(Continued)

n **Table B-1** Picture translator parameter block (Continued)

Offset		Length (bytes)		Description
12	(0x000C)	4	`dataHandle`	This placeholder can be used by the translator. For example, some compilers do not allow global variables in a code resource, and this field could hold a handle that contains all necessary global variables. If that were the case, the translator would be responsible for allocating and disposing of the handle.
16	(0x0010)	74	`theReply`	Standard reply record. This record is set up to describe the file being translated.
90	(0x005A)		`thisTranslator`	A TransDescribe record (see `XTNDTextTranslator.h`). This is the translator description for the translator being called.

Table B-2 depicts the import translator parameter block.

n **Table B-2** Import translator parameter block

Offset		Length (bytes)		Description
0	(0x0000)	4	`textBuffer`	Pointer to 256 bytes of data.
4	(0x0004)	2	`directive`	Indicates the desired action to be taken by the translator. Also used by the translator to return its response to the application.
6	(0x0006)	2	`Error`	Return a 0 here if successful, or a number representing the error.
8	(0x0008)	4	`textLength`	The number of characters of information the translator is returning.

(Continued)

n **Table B-2** Import translator parameter block (Continued)

Offset		Length (bytes)		Description
12	(0x000C)	2	`translatorState`	Available for translator use; typically used to keep track of location use in the document.
14	(0x000E)	2	`refNum`	Reference number of the fork currently being read by the translator.
16	(0x0010)	2	`txtFace`	Current text face. (Values are defined in the header `XTNDTextTranslator.h`.)
18	(0x0012)	2	`txtSize`	Current font size.
20	(0x0014)	2	`txtFont`	Font family number.
22	(0x0016)	2	`txtColor`	XTND 1.3 color value. (Note that these are different from QuickDraw values.) Value Meaning 0 White 1 Black 2 Red 3 Green 4 Blue 5 Cyan 6 Magenta 7 Yellow 8-255 reserved
24	(0x0018)	2	`txtJust`	Justification value of the text. Values are as follows: Value Meaning 0 Left 1 Center 2 Right 3 Justified

(Continued)

n **Table B-2** Import translator parameter block (Continued)

Offset		Length (bytes)	Description	
26	(0x001A)	2	`unused1`	This value should always be 0.
28	(0x001C)	4	`paraFmts`	A pointer to this paragraph's format array. The paragraph format record is defined in the section "Data Types and Conventions" in Chapter 1.
32	(0x0020)	4	`tabs`	A pointer to the array of tabs for this paragraph. The *TabSpec* format is defined in the section "Data Types and Conventions" in Chapter 1.
36	(0x0024)	1	`unused2`	A currently unused Boolean.
37	(0x0025)	1	`numCols`	Number of columns in the document.
38	(0x0026)	2	`currentStory`	The story currently being read. (The stories are defined in `XTNDTextTranslator.h`.) Value Meaning 1 rightHeaderStory 2 leftHeaderStory 3 headerStory 4 rightFooterStory 5 leftFooterStory 6 footerStory 7 footnoteStory 8 mainStory
40	(0x0028)	4	`miscData`	A long word that is used in the importing of pictures and other special characters.
44	(0x002C)	2	`storyHeight`	Height of the story. If `storyHeight` is set to 0, the application should calculate the correct height for the story.
46	(0x002E)	1	`decimalChar`	The default character on which to align a decimal tab.

(Continued)

n **Table B-2** Import translator parameter block (Continued)

	Offset	Length (bytes)	Description	
47	(0x002F)	1	`autoHyphenate`	TRUE if this document is automatically hyphenated.
48	(0x0030)	4	`printRecord`	A handle to the print record. This is initialized to nil — if your document is associated with a print record, you must create the handle and pass that back.
52	(0x0034)	4	`topMargin`	(Fixed) The top page margin (in points). Default is 72 (0x00480000), which is 1 inch.
56	(0x0038)	4	`bottomMargin`	(Fixed) The bottom page margin (in points). Default is 72 (0x00480000), which is 1 inch.
60	(0x003C)	4	`leftMargin`	(Fixed) The left page margin (in points). Default is 72 (0x00480000), which is 1 inch.
64	(0x0040)	4	`rightMargin`	(Fixed) The right page margin (in points). Default is 72 (0x00480000), which is 1 inch.
68	(0x0044)	4	`gutter`	(Fixed) The space between columns (in points). Possible values are 3 (0x00030000) through 288 (0x0120000) (4 inches). Default value is 0x000C0000.
72	(0x0048)	2	`startPageNum`	Starting Page Number.
74	(0x004A)	2	`startFootnoteNum`	Starting Footnote number.
76	(0x004C)	4	`footnoteText`	This is a pointer to a Pascal-type string of up to 9 characters. The first character is a length byte. If the length byte is 0, then an auto footnote is assumed.
80	(0x0050)	1	`rulerShowing`	TRUE if the ruler is to be shown.
81	(0x0051)	1	`doubleSided`	TRUE if document is to have left / right pages.

(Continued)

n **Table B-2** Import translator parameter block (Continued)

Offset		Length (bytes)	Description	
82	(0x0052)	1	`titlePage`	TRUE if document is to have a title page.
83	(0x0053)	1	`endnotes`	TRUE if footnotes are to be displayed as endnotes.
84	(0x0054)	1	`showInvisibles`	TRUE if invisible characters are to be shown.
85	(0x0055)	1	`showPageGuides`	TRUE if page guides are to be shown.
86	(0x0056)	1	`showPictures`	TRUE if pictures are to be shown.
87	(0x0057)	1	`autoFootnotes`	TRUE if footnotes are to be numbered automatically.
88	(0x0058)	4	`pagePoint`	Use this value if your document's page character is placed at a QuickDraw point offset from the top of the header or footer.
92	(0x005C)	4	`datePoint`	Use this value if your document's date character is placed at a QuickDraw point offset from the top of the header or footer.
96	(0x0060)	4	`timePoint`	Use this value if your document's time character is placed at a QuickDraw point offset from the top of the header or footer.
100	(0x0064)	4	`globalHandle`	A place to store your global variables. You have to create the handle and ensure that you dispose of it when you are through. The application will not use, change, or dispose of this handle. While THINK C allows you to create a code resource that can have global variables, others may not have this luxury.

104	(0x0068)	1	`smartQuotes`	TRUE if smart quotes feature (automatically placed “curly” quotation marks) is to be turned on.

(Continued)

Table B-2 Import translator parameter block (Continued)

Offset		Length (bytes)	Description	
105	(0x0069)	1	`fractCharWidths`	TRUE if fractional character widths are to be turned on.
106	(0x006A)	2	`hRes`	The horizontal resolution of the document. Default is 72.
108	(0x006C)	2	`vRes`	The vertical resolution of the document. Default is 72.
112	(0x0070)	8	`windowRect`	A standard QuickDraw rectangle representing the document window. If you do not change this field, the window will be placed in the default location.
120	(0x0078)	74	`theReply`	Standard reply record. This record is set up to describe the file being translated.
194	(0x00C2)	216	`thisTranslator`	A TransDescribe record (see `XTNDTextTranslator.h`). This is the translator description for the translator being called.

Table B-3 depicts the export translator parameter block.

Table B-3 Export translator parameter block

Offset		Length (bytes)		Description
0	(0x0000)	2	`directive`	Indicates the desired action to be taken by the translator. Also used by the translator to return its response to the application. This field, unlike the `directive` field in the import parameter block, is defined as an enumeration, which is not the same as a short. Since the enumeration contains fewer than 256 elements, it uses only one byte of this field, and the other byte is a filler to maintain word alignment of the surrounding fields.
2	(0x0002)	4	`result`	A pointer to a short integer, which should contain 0 if there is no error.
6	(0x0006)	4	`refNum`	A pointer to a short integer. This variable contains the file reference number of the open fork currently being written to.
10	(0x000A)	4	`textLength`	A pointer to a long word representing the number of characters being exported.
14	(0x000E)	4	`globalHandle`	A place to store your global variables. You have to create the handle and ensure that you dispose of it when you are through. The application will *not* use, change, or dispose of this handle.
18	(0x000E)	4	`reserved1`	Do not adjust these values.

(Continued)

Table B-3 Export translator parameter block (Continued)

	Offset	Length (bytes)	Description	
22	(0x0016)	4	`textBuffer`	A handle to the text being exported.
26	(0x001A)	4	`txtFace`	A pointer to a short word describing the current text face.
30	(0x001E)	4	`txtSize`	A pointer to a short word describing the current text size.
34	(0x0022)	4	`txtFont`	A pointer to a short word containing the current font family ID.
38	(0x0026)	4	`txtColor`	A pointer to a byte describing the MacWrite II color of the text.
42	(0x002A)	4	`txtJust`	A pointer to a short word describing the justification of the text.
46	(0x002D)	4	`paraFmts`	A pointer to a paragraph format array.
50	(0x0032)	4	`tabs`	A pointer to a tab-specification array.
54	(0x0036)	4	`thePicture`	A QuickDraw PicHandle.
58	(0x003A)	8	`pictRect`	A rectangle describing the display rectangle of the above picture.
66	(0x0042)	2	`headerStatus`	
68	(0x0044)	2	`footerStatus`	Fields to show if headers or footers appear on the right page,on the left page, or on every page. (described later in this table.)
70	(0x0046)	2	`currentStory`	The current story.
72	(0x0048)	2	`numCols`	The number of columns in the document.
74	(0x004A)	4	`topMargin`	(Fixed) The top page margin (in points).
78	(0x004D)	4	`bottomMargin`	(Fixed) The bottom page margin (in points).
82	(0x0052)	4	`leftMargin`	(Fixed) The left page margin (in points).

(Continued)

n **Table B-3** Export translator parameter block (Continued)

Offset		Length (bytes)	Description	
86	(0x0056)	4	`rightMargin`	(Fixed) The right page margin (in points).
90	(0x005A)	4	`gutter`	(Fixed) The space between columns (in points). Possible values are 3 (0x00030000) through 288 (0x0120000, or 4 inches).
94	(0x005D)	4	`totalCharCount`	Total number of characters in document.
98	(0x0062)	4	`footnoteOffset`	Footnote offset into document.
102	(0x0066)	4	`footnoteText`	Pointer to a Pascal string that contains the text for a footnote marker, if it is not an automatic footnote.
106	(0x006A)	2	`startPageNum`	Starting page number.
108	(0x006C)	2	`startFootnoteNum`	Starting footnote number.
110	(0x006E)	1	`rulerShowing`	TRUE if ruler is showing.
111	(0x006F)	1	`doubleSided`	TRUE if document contains left/right pages.
112	(0x0070)	1	`titlePage`	TRUE if document has a title page.
113	(0x0071)	1	`endnotes`	TRUE if the user has opted to put footnotes at the end of the document.
114	(0x0072)	1	`showInvisibles`	TRUE if the document has invisible elements.
115	(0x0073)	1	`showPageGuides`	TRUE if the page guides are visible.
116	(0x0074)	1	`showPictures`	TRUE if pictures are displayed.
117	(0x0075)	1	`autoFootnotes`	TRUE if footnotes are numbered automatically.
118	(0x0076)	1	`footnotesExist`	TRUE if footnotes are being exported.

(Continued)

n **Table B-3** Export translator parameter block (Continued)

Offset	Length (bytes)	Description	
120 (0x0078)	4	`printRecord`	This document's `THPrint`.
124 (0x007C)	4	`pagePoint`	Use this value if your document's page character is placed at a QuickDraw point offset from the top of the header or footer.
128 (0x0080)	4	`datePoint`	Use this value if your document's time character is placed at a QuickDraw point offset from the top of the header or footer.
132 (0x0084)	4	`timePoint`	Use this value if your document's time character is placed at a QuickDraw point offset from the top of the header or footer.
136 (0x0088)	1	`smartQuotes`	TRUE if smart quotes feature (automatically placed "curly" quotation marks) is turned on.
137 (0x0089)	1	`fractCharWidths`	TRUE if fractional character widths are turned on.
138 (0x008A)	2	`hRes`	The horizontal resolution of the document. Default is 72.
140 (0x008C)	2	`vRes`	The vertical resolution of the document. Default is 72.
142 (0x008E)	8	`windowRect`	A standard QuickDraw rectangle representing the document window. If you change this field, the window is placed in the default location.
150 (0x0096)	74	`theReply`	Standard reply record. This record is set up to describe the file being translated.
224 (0X00E0)	216	`thisTranslator`	A `TransDescribe` (see `XTNDTestTranslator.h`). This is the translator description for the translator being called.

Appendix C Header samples

Appendix C contains import and export directives from the C header file `XTNDTextTranslator.h`.

Import directives

```
enum
{
   importAcknowledge = -1,/*        Set by translator to acknowledge an
                                    action                              */

   importGetResources,
   importInitAll,

   importInitRightHeader,
   importInitLeftHeader,
   importInitHeader,

   importInitRightFooter,
   importInitLeftFooter,
   importInitFooter,

   importInitMain,

   importInitFootnote,

   importGetText,

   importCloseRightHeader,
   importCloseLeftHeader,
   importCloseHeader,

   importCloseRightFooter,
   importCloseLeftFooter,
   importCloseFooter,

   importCloseMain,

   importCloseFootnote,

   importCloseAll
};
```

Export directives

```
enum Directives
{
   exportAcknowledge = -1,          /*    Translator acknowledges
                                          directives */

   exportInit,

   exportOpenRightHeader,
   exportOpenLeftHeader,
   exportOpenHeader,

   exportOpenRightFooter,
   exportOpenLeftFooter,
   exportOpenFooter,

   exportOpenFootnote,
   exportOpenMain,

   exportWriteText,

   exportCloseRightHeader,
   exportCloseLeftHeader,
   exportCloseHeader,

   exportCloseRightFooter,
   exportCloseLeftFooter,
   exportCloseFooter,

   exportCloseFootnote,
   exportCloseMain,

   exportCloseAll,

   exportWriteResources

};
```

Appendix D Code samples

Appendix D contains sample code for 'FDIF', picture translators, text import translators, and export translators.

'FDIF' sample code

```
void    main(TwoWayInfo, Creator)
register    short         *TwoWayInfo;
register    ResType  Creator;
{
register    short fnum = *TwoWayInfo;
   long count = 2, dBytes;

   if (Creator != 'MACA')
   {    /*  FTYP will not allow this, so we can't get here            */
        *TwoWayInfo = 0;
   This should only be for file errors */
        return;  /*   Incorrect creator, so return 0                 */
   }
   *TwoWayInfo = FSRead(fnum, &count, &dBytes);
   if (*TwoWayInfo) return;
   if (dBytes == 6L)
        *TwoWayInfo = 1;  /*   Correct version, so return 1           */
   else
        *TwoWayInfo = 2;  /*   Incorrect version, so
                          return 2 */
}
```

Picture translator sample code

```
void main(ourPtr)
register PictFltrParamBlkPtr ourPtr;
{
   long     pictSize, savePictSize;
   Handle   thePicture;

   ourPtr->Error = noErr;

   /*   only respond to directives we understand                      */
   if (ourPtr->directive != pictGetPicture) return;

   /* Determine the size of the picture (= size of the file
                                                  - 512)              */
   GetEOF(ourPtr->dataRefNum, &pictSize);
   pictSize = pictHeaderSize;
   savePictSize = pictSize;
```

```
   /*   Create the picture handle  */
   thePicture = NewHandle(pictSize);
   if (ourPtr->result = MemError()) return;

   /*   skip over the header information    */
   if (ourPtr->result = SetFPos(ourPtr->dataRefNum, fsFromStart,
pictHeaderSize)) return;

   /*   Read in the picture    */
   if (ourPtr->result = FSRead(ourPtr->dataRefNum, &pictSize,
*thePicture)) return;
   if (savePictSize != pictSize) {
        DisposHandle(thePicture);
        ourPtr->result = ioErr;
        return;
   }
   /*   We have successfully read in a picture!!                    */
   ourPtr->thePicture = (PicHandle) thePicture;
}
```

Text import translator sample code

```
void main(importParms)
ImportParmBlkPtr importParms;
{
   switch(importParms->directive) {

        case importGetResources:
             MacWriteGetResources();
             break;

        case importInitAll:
             InitMacWrite();
             break;

        case import_INIT_HEADER:
             InitHeaders();
             break;

        case import_INIT_FOOTER:
             InitFooters();
             break;
```

```
        case importInitMain:
             InitMain();
             break;

        case importGetText:
             GetText();
             break;

        case importCloseAll:
             MacWriteCleanUp();
             break;

      /*    These calls are not recognized by this
                  translator  */
        case importInitRightHeader:
        case importInitLeftHeader:
        case importInitRightFooter:
        case importInitLeftFooter:
        case importInitFootnote:
        case importCloseRightHeader:
        case importCloseLeftHeader:
        case importCloseHeader:
        case importCloseMain:
        case importCloseFootnote:
             break;
}
```

Export translator sample code

```
void main(exportParms)
ExportParmBlkPtr exportParms;
{
   switch (exportParms->directive)
   {
        case exportInit:
             InitMacWrite();
             break;

        case exportOpenRightHeader:
      case exportOpenHeader:  /*    cannot have both of
                                          these    */
             InitHeader();
             break;

        case exportOpenLeftHeader:
        /*   Only accept left headers if there are no right headers    */
             if (!(exportParms->headerStatus & rightPage))
                  InitHeader();
             break;

        case exportOpenRightFooter:
      case exportOpenFooter:  /*    cannot have both of
                                          these    */
             InitFooter();
             break;

        case exportOpenLeftFooter:
        /*   Only accept left footers if there are no right footers    */
             if (!(exportParms->footerStatus & rightPage))
                  InitFooter();
             break;

        case exportOpenMain:
             InitMain();
             break;

        case exportWriteText:
             WriteText();
             break;

        case exportCloseRightHeader:
        case exportCloseLeftHeader:
```

```
        case exportCloseHeader:
        case exportCloseRightFooter:
        case exportCloseLeftFooter:
        case exportCloseFooter:
        case exportCloseMain:
             CloseStory();
             break;

        case exportCloseAll:
             TidyUp();
             break;

        case exportWriteResources:
             WriteResources();
             break;
   }
}
```