Technical Note TN1181

Sherlock's Find by Content Text Extractor Plug-ins

CONTENTS

This Technote describes the API for creating Find By Content Text Extractor Plug-ins. Text Extractor Plug-ins are used by Find by Content to extract the textual information stored in a document when it is creating indexes and summarizing files. By doing so, it is possible for users to avoid indexing peripheral data such as formatting commands, HTML tags, and other data that does not relate to the information stored in the document. By creating Text Extractor Plug-ins for their document types, developers make it possible for users to conduct meaningful searches for information stored in documents created by their applications.

Text Extractor Plug-ins can be created for use with Mac OS 8.6 and later. Mac OS 8.6 was shipped with two Text Extractor Plug-ins: the "HTML Text Extractor" and the "PDF Text Extractor." The "HTML Text Extractor" strips the HTML tags from HTML files and returns the text stored therein; the "PDF Text Extractor" returns the textual information from Adobe®'s Portable Document Format (PDF) files. In Mac OS 8.5, indexing HTML files meant that both the text stored in the document and the HTML tags were incorporated into indexes. Furthermore, PDF files were excluded from the indexing process. In Mac OS 8.6, meaningful textual information extracted from these files is incorporated into index files used by Find By Content.

This Technote provides information necessary for creating and installing Text Extractor Plug-ins. In addition, an annotated example

Text Extractor Plug-in is provided. Developers can easily modify this example to create their own plug-in for use with their own file formats.

Updated: [Jan 18 2000]

Overview

Text Extractors improve the accuracy of indexing and summarizing files. As an example, consider the HTML file shown in Listing 1.

```
<!--This HTML file contains both HTML tags and ASCII text.  For
indexing purposes, it would be more useful to ignore the tags and only
incorporate the document's text into the index.-->

<HTML>
<BODY>
This is a sample document.
</BODY>
</HTML>
```

Listing 1. A sample HTML file.

Without knowing the HTML format, every word above would get indexed, so searching for "body" in Find by Content would find the above document, but when a user opened the file in her web browser, she would not see "body". Similarly, summarizing a HTML document would show HTML Tags in the summary. The HTML Text Extractor knows the format of an HTML file so it will skip the HTML Tags and return just the text that a user would see viewing the document. For the above example, "This is a sample document." would be the only text that is indexed.

Back to top

Text Extractor Plug-ins Defined

Text Extractor Plug-ins are Code Fragments that have the following characteristics:

- File Type: `'fbce'`
- Creator Type: `'fndf'`
- Code Fragment Name: "IATextExtractor"
- System Location: "Find by Content Plug-ins" folder of the "Find" folder of the "Extensions" folder. The folder type `kFindByContentPluginsFolderType` (`'fbcp'`) can be passed to the function `FindFolder` to locate the folder.
- Exported Functions - A Text Extractor Plug-in must implement and export all of the following functions:
    - `IAPluginInit` - When a text extractor plug in is opened, the exported function IAPluginInit is called.
    - `IAPluginTerm` - When a session with a text extractor plug-in is terminated, the function IAPluginTerm is called. At this time the plug-in can perform any needed cleanup.
    - `IAGetExtractorVersion` - Returns the version of the Text Extractor Interface that plug-in corresponds to.

- IACountSupportedDocTypes - Returns the number of document types the plug-in knows how to handle. This call returns the maximum valid index for the call IAGetIndSupportedDocType.
- IAGetIndSupportedDocType - Returns the *nth* document type the plug-in supports (first item is 1 not 0). Documents are identified by Multipurpose Internet Mail Extension (MIME) type and subtype for example an HTML document would have a MIME type of "text/html".
- IAOpenDocument - Creates a reference to the text of a document, the IADocRef type is on opaque type that it defined by the plug-in to reference the document. The IADocAccessorPtr contains a reference to a document and pointers to functions used to access the document. The document accessor pointer will be valid for all calls that use the returned IADocRef until IACloseDocument is called.
- IACloseDocument - Perform any needed cleanup for the plug-in defined IADocRef object.
- IAGetNextTextRun - Given a open document reference, get the next run of text associated with the item. Fills the buffer with the next run of text. On input ioSize is the size of the buffer, on output ioSize is the number of bytes written to the buffer. If the encoding or languages of a document changes errIAEndOfTextRun should be returned. Note: a result of errIAEndOfTextRun does not necessarily mean that the routine will return an empty buffer.
- IAGetTextRunInfo - Gets the encoding and language of the text that was returned in the last call to IAGetNextTextRun.

A Text Extractor Plug-in's resource file may contain one or more 'mimp' resources that advertise the kinds of files the plug-in is able to process. The format of these resources is defined in the Registering the MIME Types a Plug-in can Understand section below.

Back to top

Registering the MIME Types a Plug-in can Understand

Clients of Text Extractors need to map documents to a MIME type. To help clients determine the document types a plug-in understands, a plug-in can include one or more 'mimp' resources in its resource file. Definitions for defining your own 'mimp' can be found in the file "IAExtractor.r". As shown in Listing 2, 'mimp' resources contain information about file Finder types and file extensions that map to a MIME type.

```
/* An example 'mimp' resources for Portable Document Format (PDF) documents. */

#include "IAExtractor.r"
resource 'mimp' (128) {
    kIACurrentMIMEMappingVersion,
    'PDF ',                     /* file type */
    'CARO',                     /* file creator */
    ".pdf",                     /* file extension */
    "application/pdf",          /* MIME type   */
    "Portable Document Format"   /* description */
};
```

Listing 2. A sample 'mimp' resource for PDF files.

When creating indexes, Find By Content uses calls to Internet Config to discover the file's MIME type. Once a file's MIME type has been discovered, it then uses the a Text Extractor Plug-in capable of extracting text from the file (based on the MIME types the extractor advertises it can decode in its 'mimp' resource).

MIME types reported by Text Extractor Plug-ins must be of the format type "/" subtype otherwise the extractor will be ignored. Also, an extractor's initialization function should verify (and correct, if necessary) that any entries in Internet Config's file mapping database referencing the extractor's type and creator specify the same mime type as the Text Extractor Plug-in.

Back to top

Structures Used By Plug-ins

Find By Content provides a number of routines and callbacks that can be used by Text Extractor Plug-ins. These callbacks provide access to memory allocation and file input. The following sections describe the structures used by Find By Content to provide these callbacks and the callbacks themselves.

Application developers wanting to call Text Extractor Plug-ins from their own code will want to create and initialize these structures themselves. Examples of how to do this can be found later in the Calling a Text Extractor Plug-in from an Application section below.

The IAPluginInitBlock Structure

The `IAPluginInitBlock` record provides call back routines that remain constant the entire time a Text Extractor Plug-in is open. A pointer to this structure is passed as a parameter to the plug-in's `IAPluginInit` routine; and, it is safe for a plug-in to save a pointer to this structure and make callbacks through it any time before the `IAPluginTerm` routine is called. Listing 3 shows the contents of the `IAPluginInitBlock` structure and prototypes for the callbacks made available by this structure. This structure and macros for making callbacks (shown as routine prototypes for illustrative purposes in Listing 3) are defined in the file "IAExtractor.h".

```
/* IAPluginInitBlock structure definition */

typedef struct IAPluginInitBlock* IAPluginInitBlockPtr;

struct IAPluginInitBlock {
        IAAllocUPP  Alloc;
        IAFreeUPP   Free;
        IAIdleUPP   Idle;
};
typedef struct IAPluginInitBlock IAPluginInitBlock;


    /* Routine Prototypes */

void* CallIAAllocProc(IAAllocUPP Alloc, UInt32 inSize);

void CallIAFreeProc(IAFreeUPP Free, void* object);

UInt8 CallIAIdleProc(IAIdleUPP Idle);
```

Listing 3. Declaration of the `IAPluginInitBlock` structure and prototypes that can be used for calling the routines referenced in the structure.

`IAPluginInitBlock` provides callbacks for allocating memory and an `idle` callback that can be called during lengthy operations. Plug-ins should use the memory allocation routines provided in this structure instead of direct calls to the Memory Manager. Callbacks provided by this structure are described below.

CallIAAllocProc

```
void* CallIAAllocProc(
        IAAllocUPP Alloc,
        UInt32 inSize);
```

`Alloc` - the value stored in the Alloc field in the `IAPluginInitBlock` structure.

`inSize` - The number of bytes to allocate.

result - a pointer to a block of storage or `NULL` if the request cannot be allocated.

`CallIAAllocProc` is a callback procedure provided in the `IAPluginInitBlock` structure that can be called by plug-ins to allocate memory.

`CallIAAllocProc` can be used for allocating memory. Plug-ins should use this callback for all memory requests. If successful, the callback returns a pointer to a block containing the requested number of bytes. If an error occurs or there is not enough memory to complete the request, then the callback returns `NULL`.

CallIAFreeProc

```
void CallIAFreeProc(
        IAFreeUPP Free,
        void* object);
```

`Free` - the value stored in the Free field in the `IAPluginInitBlock` structure.

`object` - a pointer to a block of memory allocated through the `CallIAAllocProc` callback.

`CallIAFreeProc` is a callback procedure provided in the `IAPluginInitBlock` structure that can be called by plug-ins to release memory allocated by the `CallIAAllocProc` routine.

`CallIAFreeProc` can be used for deallocating memory allocated by calls to the `CallIAAllocProc` callback.

CallIAIdleProc

```
UInt8 CallIAIdleProc(
        IAIdleUPP Idle);
```

`Idle` - the value stored in the `idle` field in the `IAPluginInitBlock` structure

`result` - non-zero if the current operation should be canceled, zero to continue.

`CallIAIdleProc` is a callback procedure provided in the `IAPluginInitBlock` structure that can be called by plug-ins while they are processing lengthy tasks.

`CallIAIdleProc` should be called by a plug-in during lengthy tasks. By calling this routine, plug-ins can allow other tasks time to run. If this callback returns any value other than zero, then the plug-in should stop processing immediately and return a `errIACanceled` result. If the `idle` callback returns zero, then the plug-in should continue processing and, perhaps, call the `idle` procedure again if necessary.

Applications developers wanting to call Text Extractor Plug-ins from inside of their own applications will have to initialize this structure and define the necessary callbacks themselves. An example showing how to set up a `IAPluginInitBlock` structure can be found in the Setting up the `IAPluginInitBlock` structure section later in this document.

Back to top

The `IADocAccessorRecord` Structure

The `IADocAccessorRecord` provides callbacks for accessing information in files. Plug-ins should be aware that although the contents of the `IAPluginInitBlock` will remain constant during the time while a plug-in is open (between calls to `IAPluginInit` and `IAPluginTerm`), it is possible that the plug-in will be passed one or more `IADocAccessorRecord` structures that refer to different files. However, it is safe to assume that the `IADocAccessorRecord` structure passed to a plug-in's `IAOpenDocument` routine will remain the same until the plug-in's `IACloseDocument` routine is called. Listing 4 shows the definition or the `IADocAccessorRecord` and macros (shown as routine prototypes for illustrative purposes) that can be used to call back through this structure.

```
    /* IADocAccessorRecord structure definition. */

typedef struct IADocAccessorRecord*     IADocAccessorPtr;

struct IADocAccessorRecord {

            /* docAccessor is an opaque type used by Find By Content
               to track the file.  It is not possible for plug-ins to
               access this information. */

        IADocAccessorRef                docAccessor;
        IADocAccessorOpenUPP            OpenDoc;
        IADocAccessorCloseUPP           CloseDoc;
        IADocAccessorReadUPP            ReadDoc;
        IASetDocAccessorReadPositionUPP SetReadPosition;
        IAGetDocAccessorReadPositionUPP GetReadPosition;
        IAGetDocAccessorEOFUPP          GetEOF;
};
typedef struct IADocAccessorRecord IADocAccessorRecord;


    /* Routine Prototypes. */

OSStatus CallIADocumentAccessorOpen(IADocAccessorRef inAccessor);

OSStatus CallIADocumentAccessorClose(IADocAccessorRef inAccessor);

OSStatus CallIADocumentAccessorRead(IADocAccessorRef inAccessor,
        void* buffer, UInt32* ioSize);

OSStatus CallIASetDocumentAccessorReadPosition(IADocAccessorRef inAccessor,
        SInt32 inMode, SInt32 inOffset);

OSStatus CallIAGetDocumentAccessorReadPosition(IADocAccessorRef inAccessor,
        SInt32* outPostion);
```

```
OSStatus CallIAGetDocumentAccessorEOF(IADocAccessorRef inAccessor, SInt32*
outEOF);


/* macros corresponding to the routine prototypes above */

#define CallIADocumentAccessorOpen(accessor) \
InvokeIADocAccessorOpenUPP((accessor)->docAccessor, \
(accessor)->OpenDoc)

#define CallIADocumentAccessorClose(accessor) \
    InvokeIADocAccessorCloseUPP((accessor)->docAccessor,\
    (accessor)->CloseDoc)

#define CallIADocumentAccessorRead(accessor, buffer, size) \
    InvokeIADocAccessorReadUPP((accessor)->docAccessor, (buffer),\
    (size), (accessor)->ReadDoc)

#define CallIASetDocumentAccessorReadPosition(accessor, mode, offset) \
    InvokeIASetDocAccessorReadPositionUPP((accessor)->docAccessor,\
    (mode), (offset), (accessor)->SetReadPosition)

#define CallIAGetDocumentAccessorReadPosition(accessor,\
    outPosition) \
    InvokeIAGetDocAccessorReadPositionUPP((accessor)->docAccessor,\
    (outPosition), (accessor)->GetReadPosition)

#define CallIAGetDocumentAccessorEOF(accessor, outEOF) \
    InvokeIAGetDocAccessorEOFUPP((accessor)->docAccessor, \
    (outEOF), accessor)->GetEOF)
```

Listing 4. Declaration of the `IADocAccessorRecord` structure and prototypes that can be used for calling the routines referenced in the structure.

The `IADocAccessorRecord` defined in Listing 4 provides plug-ins with all the necessary resources for accessing files. Plug-ins should not make calls to the File Manager directly. Instead, they should perform all file input operations necessary for accessing a file through these callbacks. Fields and callbacks defined in this structure are discussed below.

CallIADocumentAccessorOpen

```
OSStatus CallIADocumentAccessorOpen(
        IADocAccessorRef inAccessor);
```

`inAccessor` - a pointer to the `IADocAccessorRecord` passed to the `IAOpenDocument` routine.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`CallIADocumentAccessorOpen` is a callback procedure provided in the `IADocAccessorRecord` structure that can be called by plug-ins to open a file for input.

`CallIADocumentAccessorOpen` opens the document for reading. Plug-ins should call this routine to open the document for reading before making any of the input calls described below.

CallIADocumentAccessorClose

```
OSStatus CallIADocumentAccessorClose(
        IADocAccessorRef inAccessor);
```

`inAccessor` - a pointer to the `IADocAccessorRecord` passed to the `IAOpenDocument` routine. `inAccessor` must be in the open state when this routine is called.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`CallIADocumentAccessorClose` is a callback procedure provided in the `IADocAccessorRecord` structure that can be called by plug-ins to close a file that was opened by a call to `CallIADocumentAccessorOpen`.

`CallIADocumentAccessorClose` should be called to close a file opened by a call to `CallIADocumentAccessorOpen`.

CallIADocumentAccessorRead

```
OSStatus CallIADocumentAccessorRead(
        IADocAccessorRef inAccessor,
        void* buffer,
        UInt32* ioSize);
```

`inAccessor` - a pointer to the `IADocAccessorRecord` passed to the `IAOpenDocument` routine. `inAccessor` must be in the open state when this routine is called.

`buffer` - a pointer to a buffer where the data should be stored.

`ioSize` - a pointer to a 32-bit integer containing the number of bytes to be read. When the routine returns, this value will have been updated to the actual number of bytes read.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`CallIADocumentAccessorRead` is a callback procedure provided in the `IADocAccessorRecord` structure that can be called by plug-ins to read data from a file.

`CallIADocumentAccessorRead` reads `*ioSize` bytes from the file starting at the current read file position. On return, `*ioSize` will reflect the actual number of bytes read and the routine's result will indicate the success of the call. If this callback returns an `eofErr` error, be sure to check the value stored in `*ioSize` as it is possible that some bytes may have been read into the buffer before the end of the file was encountered. Calls to `CallIADocumentAccessorRead` advance the read position for the file past the bytes that have been read - the next call to `CallIADocumentAccessorRead` begins where the last one left off.

CallIASetDocumentAccessorReadPosition

```
OSStatus CallIASetDocumentAccessorReadPosition(
        IADocAccessorRef inAccessor,
        SInt32 inMode,
        SInt32 inOffset);
```

`inAccessor` - a pointer to the `IADocAccessorRecord` passed to the `IAOpenDocument` routine. `inAccessor` must be in the open state when this routine is called.

`inMode` - contains one of the following positioning constants:

- `kIAFromStartMode` - `inOffset` contains a value to be interpreted as an offset from the start of the file.
- `kIAFromCurrMode` - `inOffset` contains a value to be interpreted as an offset the current read position.
- `kIAFromEndMode` - `inOffset` contains a value to be interpreted as an offset from the end of the file.

`inOffset` - contains a 32-bit signed integer used to offset the current read position.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`CallIASetDocumentAccessorReadPosition` is a callback procedure provided in the `IADocAccessorRecord` structure that can be called by plug-ins to set the position where the next read will take place when `CallIADocumentAccessorRead` is called.

`CallIASetDocumentAccessorReadPosition` can be used to set the position where the next call to `CallIADocumentAccessorRead`will begin reading bytes from the file. When a file is first opened, its read position is set to the beginning of the file.

CallIAGetDocumentAccessorReadPosition

```
OSStatus CallIAGetDocumentAccessorReadPosition(
        IADocAccessorRef inAccessor,
        SInt32* outPostion);
```

`inAccessor` - a pointer to the `IADocAccessorRecord` passed to the `IAOpenDocument` routine. `inAccessor` must be in the open state when this routine is called.

`outPostion` - a pointer to a 32-bit value that is set to the current read position's offset from the beginning of the file.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`CallIAGetDocumentAccessorReadPosition` is a callback procedure provided in the `IADocAccessorRecord` structure that can be called by plug-ins to determine the position where the next read will take place when `CallIADocumentAccessorRead` is called.

`CallIAGetDocumentAccessorReadPosition` returns the location where the next read operation will take place in `*outPostion`. The value returned is an offset from the beginning of the file.

CallIAGetDocumentAccessorEOF

```
OSStatus CallIAGetDocumentAccessorEOF(
        IADocAccessorRef inAccessor,
        SInt32* outEOF);
```

`inAccessor` - a pointer to the `IADocAccessorRecord` passed to the `IAOpenDocument` routine. `inAccessor` must be in the open state when this routine is called.

`outEOF` - a pointer to a 32-bit value that is set to the number of bytes in the file.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`CallIAGetDocumentAccessorReadPosition` is a callback procedure provided in the `IADocAccessorRecord` structure that can be called by plug-ins to determine length of the input file.

`CallIAGetDocumentAccessorEOF` can be used to discover the length of a file. On return, `*outEOF` is set to the total number of bytes in the file.

Applications developers wanting to call Text Extractor Plug-ins from inside of their own applications will have to initialize this structure and define the necessary callbacks themselves. An example showing how to set up a `IADocAccessorRecord` structure can be found in the Setting up the `IADocAccessorRecord` structure section later in this document.

Back to top

Routines a Text Extractor Must Define

This section describes the routines that must be exported by all Text Extractor Plug-ins. This section provides a detailed description of each routine along with some discussion any important issues related to each routine.

IAPluginInit

```
OSStatus IAPluginInit(
        IAPluginInitBlockPtr initBlock,
        IAPluginRef *outPluginRef);
```

`initBlock` - contains a pointer to a `IAPluginInitBlock` structure.

`outPluginRef` - is a pointer to a 32-bit value that will be passed to other plug-in routines while the plug-in is open. A plug-in may set this value in its `IAPluginInit` routine and it will remain unchanged until `IAPluginTerm` is called.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`IAPluginInit` is a routine that must be provided in the plug-in's code fragment.

After the plug-in's code fragment has been prepared for execution, the plug-in's `IAPluginInit` routine is called. This routine provides an opportunity for a plug-in to perform any necessary initialization operations it may require.

The callbacks in the `IAPluginInitBlock` pointed to by the `initBlock` parameter remain valid while the plug-in is open (until `IAPluginTerm` is called) and may be called from any of the plug-in's other routines. The value stored in `*outPluginRef` is dedicated for the plug-in's use and may be used to store persistent state information that is to remain intact between calls to the plug-in (this value is not saved after the plug-in has been closed).

For an example illustrating how this routine could be implemented refer to Listing 6.

Back to top

IAPluginTerm

```
OSStatus IAPluginTerm(IAPluginRef inPluginRef);
```

`inPluginRef` - a 32-bit value dedicated for the plug-in's use. This value will be the same as the value the `*outPluginRef` parameter was set to in the `IAPluginInit` call.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`IAPluginTerm` is a routine that must be provided in the plug-in's code fragment.

Before a plug-in's Code Fragment Manager connection is closed, the plug-in's `IAPluginTerm` routine is called. This routine provides opportunity for the plug-in to perform any necessary cleanup operations required such as deallocating storage, closing resource files, et cetera. After this routine has been called, there will be no other calls made to the plug-in until the next time it is opened by a call to `IAPluginInit`.

For an example illustrating how this routine could be implemented refer to Listing 7.

Back to top

IAGetExtractorVersion

```
OSStatus IAGetExtractorVersion(
        IAPluginRef inPluginRef,
        UInt32 outPluginVersion);
```

`inPluginRef` - a 32-bit value dedicated for the plug-in's use. This value will be the same as the value the `*outPluginRef` parameter was set to in the `IAPluginInit` call.

`outPluginVersion` - a pointer to a 32-bit value. Your routine should set this value to `kIAExtractorCurrentVersion`.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`IAGetExtractorVersion` is a routine that must be provided in the plug-in's code fragment.

In this routine, a plug-in should set the value `*outPluginVersion` to the version of the Text Extractor Plug-in interface it was compiled against. The constant `kIAExtractorCurrentVersion`, defined in "IAExtractor.h", contains the current version of the Text Extractor Plug-in interface.

For an example illustrating how this routine could be implemented refer to Listing 8.

Back to top

IACountSupportedDocTypes

```
OSStatus IACountSupportedDocTypes(
        IAPluginRef inPluginRef,
        UInt32* outCount);
```

inPluginRef - a 32-bit value dedicated for the plug-in's use. This value will be the same as the value the *outPluginRef parameter was set to in the IAPluginInit call.

outCount - a pointer to a 32-bit integer. The plug-in should set this integer to the number of document types that it knows how to handle.

result - errIANoErr if the operation was successful, some other error code if the operation failed.

IACountSupportedDocTypes is a routine that must be provided in the plug-in's code fragment.

This routine should set *outCount to the number of document types the plug-in is able to handle. The value stored in *outCount is interpreted as the maximum valid index that can be provided as an index in IAGetIndSupportedDocType calls.

For an example illustrating how this routine could be implemented refer to Listing 9.

Back to top

IAGetIndSupportedDocType

```
OSStatus IAGetIndSupportedDocType(
        IAPluginRef inPluginRef,
        UInt32 inIndex,
        char** outMIMEType);
```

inPluginRef - a 32-bit value dedicated for the plug-in's use. This value will be the same as the value the *outPluginRef parameter was set to in the IAPluginInit call.

inIndex - a 32-bit integer value indicating the index of the document type to return. Index values range between 1 and the maximum index value returned by IACountSupportedDocTypes.

outMIMEType - a pointer value of type char. A plug-in should set this value to point to a string containing the MIME type string. The storage for this string belongs to the plug-in - if it was allocated by the plug-in, then the plug-in must deallocate it.

result - errIANoErr if the operation was successful, some other error code if the operation failed.

IAGetIndSupportedDocType is a routine that must be provided in the plug-in's code fragment.

The routine IAGetIndSupportedDocType sets *outMIMEType to point to a string containing the *nth* MIME type the plug-in is able to understand. Index values that may be provided in the inIndex parameter range from 1 (not zero) through the maximum value as reported by the IACountSupportedDocTypes call.

For an example illustrating how this routine could be implemented refer to Listing 10.

Back to top

IAOpenDocument

```
OSStatus IAOpenDocument(
        IAPluginRef inPluginRef,
        IADocAccessorPtr inAccessor,
        IADocRef* outDoc);
```

`inPluginRef` - a 32-bit value dedicated for the plug-in's use. This value will be the same as the value the `*outPluginRef` parameter was set to in the `IAPluginInit` call.

`inAccessor` - a pointer to a `IAPluginInitBlock` containing callbacks necessary for reading information from a file.

`outDoc` - a pointer to a 32-bit value available for the plug-in to use for storing information specific to the document. Normally plug-ins will store a pointer to necessary state variables specific to the document in this parameter.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`IAOpenDocument` is a routine that must be provided in the plug-in's code fragment.

`IAOpenDocument` is called before a plug-in is used to extract text from a new document. This routine provides opportunity for the plug-in to perform any initialization operations required before it begins reading text from a file. Any state variables or data buffers required for processing the file should be stored in a block of memory and a pointer to that block should be stored in `*outDoc`. This value will be passed to the routines `IAGetNextTextRun`, and `IAGetTextRunInfo` while the document is open, and then to `IACloseDocument` once all the required text has been extracted from the document. Both the `IAPluginInitBlock` pointed to by the `inAccessor` parameter and the value stored in `*outDoc` will remain valid until `IACloseDocument` is called.

For an example illustrating how this routine could be implemented refer to Listing 11.

Back to top

IACloseDocument

```
OSStatus IACloseDocument(
        IADocRef inDoc);
```

`inDoc` - The document reference value created by the plug-in the `IAOpenDocument` call containing state variables or data buffers required for processing the file.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`IACloseDocument` is a routine that must be provided in the plug-in's code fragment.

`IACloseDocument` is called after all textual information required has been extracted from the document. In this call, the plug-in should dispose of any state variables or buffers that were created specifically for the file referenced by the `inDoc` parameter.

For an example illustrating how this routine could be implemented refer to Listing 12.

Back to top

IAGetNextTextRun

```
OSStatus IAGetNextTextRun(
        IADocRef inDoc,
        void* buffer,
        UInt32* ioSize);
```

`inDoc` - The document reference value created by the plug-in the `IAOpenDocument` call containing state variables or data buffers required for processing the file.

`buffer` - a pointer to a block of memory.

`ioSize` - a pointer to a 32-bit integer value. when the routine is called, this value will equal the number of bytes available in the memory area pointed to by `buffer` parameter. After copying some text to this memory buffer, the plug-in should set this value to the actual number of bytes copied.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`IAGetNextTextRun` is a routine that must be provided in the plug-in's code fragment.

The `IAGetNextTextRun` routine should copy text from the document into the memory buffer pointed to by the `buffer` parameter until that buffer is full, or the plug-in runs out of text. If the language encoding changes from one language to another while text is being decoded, the plug-in mark that location in the text stream by returning the result code `errIAEndOfTextRun`.

When the plug-in reaches the end of the text in the file, it should return a result code of `noErr` and it should set `*ioSize` to zero indicating there is no more text to be read from the file.

For an example illustrating how this routine could be implemented refer to Listing 13.

Back to top

IAGetTextRunInfo

```
OSStatus IAGetTextRunInfo(
        IADocRef inDoc,
        char** outEncoding,
        char** outLanguage);
```

`inDoc` - The document reference value created by the plug-in the `IAOpenDocument` call containing state variables or data buffers required for processing the file.

`outEncoding` - a pointer to a variable to type char*. This is an optional parameter, and may be set to `NULL` if the caller is not interested in this value. The plug-in should store a pointer to a string in the variable pointed to by this parameter that contains the Internet name for the current character encoding for text being extracted from the file.

`outLanguage` - a pointer to a variable to type char*. This is an optional parameter, and may be set to `NULL` if the caller is not interested in this value. The plug-in should store a pointer to a string in the variable pointed to by this parameter that contains the language name for text being extracted from the file. The language corresponds to the internet standard defined in ISO-639.

result - `errIANoErr` if the operation was successful, some other error code if the operation failed.

`IAGetTextRunInfo` is a routine that must be provided in the plug-in's code fragment.

`IAGetTextRunInfo` returns information about the character encoding and the language of the text for the last buffer returned by `IAGetNextTextRun`.

Both parameters are optional and may or may not be present depending on the caller's requirements. If a parameter is not required, then it will be set to `NULL`.

If the plug-in allocates a pointer to a string and stores that pointer either in `*outEncoding` or in `*outLanguage`, then it is the plug-in's responsibility to deallocate that storage.

If either value is not known, the plug-in may store the value `NULL` in either `*outEncoding` or in `*outLanguage`. This value instructs the caller that the current character encoding or language is not known by the plug-in.

A pointer to a string containing the Internet name for the character encoding is returned in the `*outEncoding` parameter. Encoding is the internet name for an encoding (i.e., "iso-8859-1", "x-mac-roman", "euc-jp", ...).

For an example illustrating how this routine could be implemented refer to Listing 14.

Back to top

An Example Plug-in

The following annotated example illustrates how to create a Text Extractor Plug-in for the "text/plain" MIME type. As the function of this plug-in is to pass text from the file to the caller, its implementation is very simple. Developers can easily modify this example to extract text from their own file formats.

```
/* File: PlainTextExtractor.c

    Text Extractor plug-in example/shell.  */


        /* The file IAExtractor.h contains defines and structures
           necessary for creating a Text Extractor Plug-in. */

#include "IAExtractor.h"


        /* This constant is used in the example as a data value
           stored in the reference value maintained by the caller
           for the plug-in.  It's not necessary to create a plug-in,
           but it's useful for illustration. */
enum {
    kPlainTextExtractorRefType = 'text'
};

        /* This macro is used or verifying the reference value
           remains unchanged in the example. */

#define VerifyType(x) ((UInt32)(x)==(UInt32)kPlainTextExtractorRefType)
```

Listing 5. File header & imports for Text Extractor Plug-ins.

The only important aspect of the above is the header file being included. Here, the file "IAExtractor.h" containing the necessary constant and structure definitions is included.

```
/* IAPluginInit example implementation.*/

OSStatus IAPluginInit(
        IAPluginInitBlockPtr initBlock,
        IAPluginRef* outPluginRef) {

        /* validate parameters. */

    if (outPluginRef == NULL) return errIAParamErr;

        /* initialize the reference value. Plug-ins that
           require memory allocation should cache initBlock
           info here. */

    *outPluginRef = (IAPluginRef)kPlainTextExtractorRefType;

        /* Return with no error. */

    return errIANoErr;
}
```

Listing 6. `IAPluginInit` example.

The `IAPluginInit` is the first call made to the plug-in. During this call, the plug-in should set up any variables or tables required. Also, if the plug-in will require any of the callbacks found in the `IAPluginInitBlock` pointed to by the `initBlock` parameter later during its execution, then it should save a copy of this pointer.

Back to top

```
/* IAPluginTerm example implementation.*/

OSStatus IAPluginTerm(IAPluginRef inPluginRef) {

        /* validate parameters */

    if (!VerifyType(inPluginRef))
        return errIAParamErr;

        /* do other tear-down operations here... */

    ....

        /* Return with no error. */

    return errIANoErr;
}
```

Listing 7. `IAPluginTerm` example.

Normally, the `IAPluginTerm` routine will be used to deallocate any storage allocated by the plug-in, close any resource files, and other cleanup tasks that need to be performed.

Back to top

```
/* IAGetExtractorVersion example implementation.*/

OSStatus IAGetExtractorVersion(
        IAPluginRef inPluginRef,
        UInt32* outPluginVersion) {

        /* validate parameters */

    if (!VerifyType(inPluginRef) || !outPluginVersion )
        return errIAParamErr;

        /* set return value to the interface version
           this code was compiled with. */

    *outPluginVersion = kIAExtractorCurrentVersion;

        /* Return with no error. */

    return errIANoErr;
}
```

Listing 8. `IAGetExtractorVersion` example.

The value `kIAExtractorCurrentVersion` will always contain the current version for the declarations included in the file `"IAExtractor.h"`. For the current implementation this value is set to `kIAExtractorVersion1`.

Back to top

```
/* IACountSupportedDocTypes example implementation.*/

OSStatus IACountSupportedDocTypes(
        IAPluginRef inPluginRef,
        UInt32* outCount) {

        /* validate parameters*/

    if (!VerifyType(inPluginRef) || ! outCount)
        return errIAParamErr;

        /* count is max value to be passed to
           IAGetIndSupportedDocType as index */

    *outCount = 1;

        /* Return with no error. */

    return errIANoErr;
}
```

Listing 9. IACountSupportedDocTypes example.

In this example, we only support one document type - plain text documents.

Back to top

```
/* IAGetIndSupportedDocType example implementation.*/

OSStatus IAGetIndSupportedDocType(
        IAPluginRef inPluginRef,
        UInt32 inIndex,
        char **outMIMEType) {

        /* set up local variables */

    static char* supportedDocType = "text/plain";

        /* validate parameters */

    if (!VerifyType(inPluginRef) || !outMIMEType || inIndex != 1)
        return errIAParamErr;

        /* set return value */

    *outMIMEType = supportedDocType;

        /* return successfully */

    return errIANoErr;
}
```

Listing 10. IAGetIndSupportedDocType example.

In the above declaration of IAGetIndSupportedDocType the MIME type string is stored as a static variable among the plug-in's globals.

Back to top

```
/* IAOpenDocument example implementation.*/

OSStatus IAOpenDocument(
        IAPluginRef inPluginRef,
        IADocAccessorPtr inDocAccessor,
        IADocRef* outDoc) {

        /* local variables */

    OSStatus err;

        /* verify parameters */

    if (!VerifyType(inPluginRef) || !inDocAccessor || !outDoc)
        return errIAParamErr;

        /* call our opening routine */

    err = CallIADocumentAccessorOpen(inDocAccessor);
    if (err != errIANoErr)
        return err;

        /* IADocRef is defined by plug-in, in our case we are just
           reading directly from the accessor so we are defining the
           opaque type IADocRef to be an IADocAccessorPtr. */

    *outDoc = (IADocRef)inDocAccessor;

        /* return successfully */

    return errIANoErr;
}
```

Listing 11. `IAOpenDocument` example.

In the `IAOpenDocument` call shown above, the plug-in calls back through the `IAPluginInitBlock` record pointed to by the `inDocAccessor` parameter and before caching a copy of `inDocAccessor` in the document reference parameter (`*outDoc`). This value is used to refer to the document in the next few listings.

Back to top

```
/* IACloseDocument example implementation.*/

OSStatus IACloseDocument(IADocRef inDoc) {

        /* local variables */

    IADocAccessorPtr    docAccessor;
    OSStatus            err;

        /* verify parameters */

    if (inDoc == NULL)
        return errIAParamErr;

        /* Cast IADocRef to what we defined it to be in
           IAOpenDocument in this case a IADocAccessorPtr */

    docAccessor = (IADocAccessorPtr)inDoc;

        /* use the callback to close the file */

    err = CallIADocumentAccessorClose(docAc
cessor);

        /* return status of last close */

    return err;
}
```

Listing 12. `IACloseDocument` example.

In the `IACloseDocument` call shown above, the plug-in calls back through the `IAPluginInitBlock` structure to close the file. The pointer to the `IAPluginInitBlock` structure is coerced from the `inDoc` parameter where a copy was saved during the `IAOpenDocument` call shown in Listing 11.

Back to top

```
/* IAGetNextTextRun example implementation. */

OSStatus IAGetNextTextRun(
        IADocRef inDoc,
        void* buffer,
        UInt32* size) {

        /* local variables */

    IADocAccessorPtr    docAccessor;
    OSStatus            err;

        /* verify parameters */

    if (!inDoc)
        return errIAParamErr;

        /* Cast IADocRef to what we defined it to be
           in IAOpenDocument (in this case a IADocAccessorPtr). */

    docAccessor = (IADocAccessorPtr)inDoc;

        /* callback to read from the file. */

    err = CallIADocumentAccessorRead(docAcce
ssor, buffer, size);

        /* return result o read operation */

    return err;
}
```

Listing 13. `IAGetNextTextRun` example.

In the `IACloseDocument` call shown above, the plug-in calls back through the `IAPluginInitBlock` structure to read data bytes from the file. The pointer to the `IAPluginInitBlock` structure is coerced from the `inDoc` parameter where a copy was saved during the `IAOpenDocument` call shown in Listing 11.

Back to top

```
/* IAGetTextRunInfo example implementation.  */

OSStatus IAGetTextRunInfo(
        IADocRef inDoc,
        char** outEncoding,
        char** outLanguage) {

        /* we don't know the encoding or language of the file so
           set to NULL. */

    if (outEncoding != NULL) *outEncoding = NULL;
    if (outLanguage != NULL) *outLanguage = NULL;

        /* local variables */

    return errIANoErr;
}
```

Listing 14. IAGetTextRunInfo example.

In this example, we return `NULL`, indicating that both the text encoding and the language are unknown.

Back to top

Calling a Text Extractor Plug-in from an Application

Following is an example of how a client may use a Text Extractor Plug-in to extract the text of a document. Applications may use these routines or some variant of them to call Text Extractor Plug-ins to extract text from virtually any document type.

The steps below show how to set up the plug-in's code fragment, set up the callback structures, and finally how to call the plug-in to perform the text extraction. This example does not show how to find or determine the correct plug-in for a particular document.

Setting up a Text Extractor Plug-in

First, we begin by setting up the plug-in's code fragment for execution and storing pointers to the routines we want to call in a structure we will use to access the plug-in. Listing 15 contains the routines and declarations used to perform this task.

```
    /* The following typedefs correspond to the routines
    exported by Text Extractor Plug-ins.  In this example,
    we use these for calling the plug-in from our code. */

typedef OSStatus (*PluginInitCallPtr)(
    IAPluginInitBlockPtr initBlock,
    IAPluginRef* outPluginRef);

typedef OSStatus (*PluginTermCallPtr)(
        IAPluginRef inPluginRef);
```

```
typedef OSStatus (*GetExtractorVersionCallPtr)(
    IAPluginRef inPluginRef,
    UInt32* outPluginVersion);

typedef OSStatus (*CountSupportedDocTypesCallPtr)(
    IAPluginRef inPluginRef,
    UInt32* outCount);

typedef OSStatus (*GetIndSupportedDocTypeCallPtr)(
    IAPluginRef inPluginRef,
    UInt32 inIndex,
    char** outMIMEType);

typedef OSStatus (*OpenDocumentCallPtr)(
    IAPluginRef inPluginRef,
    IADocAccessorPtr inDoc,
    IADocRef* outDoc);

typedef OSStatus (*CloseDocumentCallPtr)(IADocRef inDoc);

typedef OSStatus (*GetTextRunInfoCallPtr)(IADocRef inDoc,
    char** outEncoding,
    char** outLanguage);

typedef OSStatus (*GetNextTextRunCallPtr)(
    IADocRef inDoc,
    void* buffer,
    UInt32* size);



    /* ExtractorRec is used for storing information about the
    plug-in's code fragment itself.  it contains pointers
    to the fragment's routines, and the fragment's CFM
    connection id number. */

typedef struct {
    CFragConnectionID connID;
    PluginInitCallPtr PluginInit;
    PluginTermCallPtr PluginTerm;
    GetExtractorVersionCallPtr GetExtractorVersion;
    CountSupportedDocTypesCallPtr CountSupportedDocTypes;
    GetIndSupportedDocTypeCallPtr GetIndSupportedDocType;
    OpenDocumentCallPtr OpenDocument;
    CloseDocumentCallPtr CloseDocument;
    GetNextTextRunCallPtr  GetNextTextRun;
    GetTextRunInfoCallPtr GetTextRunInfo;
} ExtractorRec, *ExtractorRecPtr;



    /* OpenExtractor loads the code fragment belonging
    to the Text Extractor Plug-in referred to by the file
    system specification record referred to by its spec
    parameter.  If successful, it returns a pointer to
    a structure containing pointers to the plug-in's
    routines.  */

static OSStatus OpenExtractor(FSSpec *spec, ExtractorRecPtr *extractor) {
    ExtractorRecPtr extr;
    Str63 fragName;
    Ptr mainAddr;
    Str255 errName;
    Boolean fragmentExists; /* tracks contents of fragConnID */
    CFragConnectionID fragConnID;
    CFragSymbolClass symbolClass;
    OSStatus err;

        /* set up locals to a known state */

    extr = NULL;
    fragmentExists = false;

        /* allocate the storage for saving information about
        the plug-in. */

    extr = (ExtractorRecPtr) NewPtrClear(sizeof(ExtractorRec));
    if (extr == NULL) { err = memFullErr; goto bail; }

        /* set up the plug-in's code fragment for use. */

    err = GetDiskFragment(spec, 0, kCFragGoesToEOF,
        fragName, kPrivateCFragCopy,
        &fragConnID, &mainAddr, errName);
    if (err != noErr) goto bail;
    fragmentExists = true;
    extr->connID = fragConnID;

        /* save pointers to the routines we want to call.  */

    err = FindSymbol(fragConnID, "\pIAPluginInit",
        (Ptr*) &extr->PluginInit, &symbolClass);
    if (err != noErr) goto bail;

    err = FindSymbol(fragConnID, "\pIAPluginTerm",
        (Ptr*) &extr->PluginTerm, &symbolClass);
    if (err != noErr) goto bail;

    err = FindSymbol(fragConnID, "\pIAGetExtractorVersion",
        (Ptr*) &extr->GetExtractorVersion, &symbolClass);
    if (err != noErr) goto bail;

    err = FindSymbol(fragConnID, "\pIACountSupportedDocTypes",
        (Ptr*) &extr->CountSupportedDocTypes, &symbolClass);
    if (err != noErr) goto bail;

    err = FindSymbol(fragConnID, "\pIAGetIndSupportedDocType",
        (Ptr*) &extr->GetIndSupportedDocType, &symbolClass);
    if (err != noErr) goto bail;
    err = FindSymbol(fragConnID, "\pIAOpenDocument",
        (Ptr*) &extr->OpenDocument, &symbolClass);
    if (err != noErr) goto bail;

    err = FindSymbol(fragConnID, "\pIACloseDocument",
        (Ptr*) &extr->CloseDocument, &symbolClass);
    if (err != noErr) goto bail;

    err = FindSymbol(fragConnID, "\pIAGetNextTextRun",
        (Ptr*) &extr->GetNextTextRun, &symbolClass);
    if (err != noErr) goto bail;

    err = FindSymbol(fragConnID, "\pIAGetTextRunInfo",
        (Ptr*) &extr->GetTextRunInfo, &symbolClass);
    if (err != noErr) goto bail;

           /* return successfully */
    *extractor = extr;
    return noErr;

bail:

    if (fragmentExists) CloseConnection(&fragConnID);
    if (extr != NULL) DisposePtr((Ptr) extr);
    return err;
}



    /* CloseExtractor unloads the plug-in's code fragment and
    releases storage allocated when it was opened. */

static void CloseExtractor(ExtractorRecPtr extr) {

        /* close the code fragment manager connection to
        the plug-in's file. */

    CloseConnection(&extr->connID);

        /* release the memory we were using to track the
        plug-in's code fragment. */

    DisposePtr((Ptr) extr);
}
```

Listing 15. Routines for setting up a Text Extractor Plug-in's code fragment for execution.

The prototypes provided in Listing 15 allow us to call back to the plug-in. Pointers to these routines are stored in the `ExtractorRec` structure.

Back to top

Setting up the `IAPluginInitBlock` structure

Routines for setting up a `IAPluginInitBlock` structure are provided in Listing 16. Here, callbacks used by the plug-in are referenced in the structure saving routine descriptors referring to them in the structure.

```
    /* routines exported in the IAPluginInitBlock record.
    Here we have defined our own set of routines that
    call through to the Mac OS memory manager. */

static void* MyIAAlloc(UInt32 inSize) {
    return (void*) NewPtr(inSize);
}

static void MyIAFreeProc(void* object) {
    DisposePtr((Ptr) object);
}

static UInt8 MyIAIdleProc(void) {
    return 0;
}



    /* NewIAPluginInitBlock allocates a new init block
    record containing memory allocation routines
    and idle routines that can be called
    by a plug-in.   If an error occurs, the function
    returns NULL.  */

static OSStatus NewIAPluginInitBlock(IAPluginInitBlockPtr *iapBlock) {
    IAPluginInitBlockPtr iBlock;
    OSStatus err;

    iBlock = NULL;

    iBlock = (IAPluginInitBlockPtr) NewPtrClear(sizeof(IAPluginInitBlock));
    if (iBlock == NULL) { err = memFullErr; goto bail; }

    iBlock->Alloc = NewIAAllocProc(MyIAAlloc);
    if (iBlock->Alloc == NULL) { err = memFullErr; goto bail; }

    iBlock->Free = NewIAFreeProc(MyIAFreeProc);
    if (iBlock->Free == NULL) { err = memFullErr; goto bail; }

    iBlock->Idle = NewIAIdleProc(MyIAIdleProc);
    if (iBlock->Idle == NULL) { err = memFullErr; goto bail; }

    *iapBlock = iBlock;
    return noErr;

bail:
    if (iBlock != NULL) {
        if (iBlock->Alloc != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) iBlock->Alloc);
        if (iBlock->Free != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) iBlock->Free);
        if (iBlock->Idle != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) iBlock->Idle);
        DisposePtr((Ptr) iBlock);
    }
    return err;
}




    /* DisposeIAPluginInitBlock releases the memory occupied
    by the init block record allocated in NewIAPluginInitBlock.   */

static void DisposeIAPluginInitBlock(IAPluginInitBlockPtr iBlock) {
    DisposeRoutineDescriptor((UniversalProcPtr) iBlock->Alloc);
    DisposeRoutineDescriptor((UniversalProcPtr) iBlock->Free);
    DisposeRoutineDescriptor((UniversalProcPtr) iBlock->Idle);
    DisposePtr((Ptr) iBlock);
}
```

Listing 16. Routines for allocating and initializing an `IAPluginInitBlock` structure.

The routines provided in Listing 16 allocate and deallocate the `IAPluginInitBlock` structure to use routines that call the Memory Manager.

Back to top

Setting up the `IADocAccessorRecord` structure

The routines and declarations provided in Listing 17 illustrate how to set up the file access callbacks for a plug-in. Here, we allocate the callback structure and another structure for keeping track off the file itself.

```
    /* MyDocumentReference contains information used by
    the caller to track the input source being used
    by the plug-in.  In this example, we are using a
    Mac OS file.  A pointer to this structure will be
    passed back to our file io routines. */

typedef struct {
    FSSpec spec;  /* a copy of the file specification record */
    Boolean docOpen; /* true when document is open */
    short refnum; /* file reference number */
} MyDocumentReference, *MyDocRefPtr;
```

```
    /* in this example, we will fill the fields of the
    IADocAccessorRecord with routine descriptors referring
    to routines that call through to the Mac OS file system.
    These routines are defined below.  */

static OSStatus MyIADocAccessorOpenProc(
            IADocAccessorRef inAccessor) {
    MyDocRefPtr refptr;
    OSErr err;

    refptr = (MyDocRefPtr) inAccessor;

    err = FSpOpenDF(&refptr->spec, fsRdPerm, &refptr->refnum);
    if (err == noErr)
        refptr->docOpen = true;
    return (OSStatus) err;
}

static OSStatus MyIADocAccessorCloseProc(
            IADocAccessorRef inAccessor) {
    MyDocRefPtr refptr;

    refptr = (MyDocRefPtr) inAccessor;

    if ( ! refptr->docOpen)
        return errIAParamErr;
    FSClose(refptr->refnum);
    refptr->docOpen = false;
    return errIANoErr;
}

static OSStatus MyIADocAccessorReadProc(
            IADocAccessorRef inAccessor,
            void* buffer, UInt32* ioSize) {
    MyDocRefPtr refptr;
    OSErr err;

    refptr = (MyDocRefPtr) inAccessor;

    if ( ! refptr->docOpen)
        return errIAParamErr;

        /* read the data */
    err = FSRead(refptr->refnum, (long *) ioSize, buffer);

        /* special case for end of file errors */
    if (err == eofErr && *ioSize != 0) err = noErr;

    return (OSStatus) err;
}

static OSStatus MyIASetDocAccessorReadPositionProc(
               IADocAccessorRef inAccessor,
               SInt32 inMode, SInt32 inOffset) {
```

```
    MyDocRefPtr refptr;
    OSErr err;

    refptr = (MyDocRefPtr) inAccessor;

    if ( ! refptr->docOpen)
        return errIAParamErr;

    switch (inMode) {
        case kIAFromStartMode:
            err = SetFPos(refptr->refnum, fsFromStart, inOffset);
            break;
        case kIAFromCurrMode:
            err = SetFPos(refptr->refnum, fsFromMark, inOffset);
            break;
        case kIAFromEndMode:
            err = SetFPos(refptr->refnum, fsFromLEOF, inOffset);
            break;
        default:
            err = errIAParamErr;
            break;
    }

    return (OSStatus) err;
}

static OSStatus MyIAGetDocAccessorReadPositionProc(
            IADocAccessorRef inAccessor,
            SInt32* outPostion) {
    MyDocRefPtr refptr;
    OSErr err;

    refptr = (MyDocRefPtr) inAccessor;

    if ( ! refptr->docOpen)
        return errIAParamErr;

    err = GetFPos(refptr->refnum, outPostion);

    return (OSStatus) err;
}

static OSStatus MyIAGetDocAccessorEOFProc(
            IADocAccessorRef inAccessor, SInt32* outEOF) {
    MyDocRefPtr refptr;
    OSErr err;

    refptr = (MyDocRefPtr) inAccessor;

    if ( ! refptr->docOpen)
        return errIAParamErr;

    err = GetEOF(refptr->refnum, outEOF);

    return (OSStatus) err;
```

```
}




    /* NewIADocAccessorRec initializes a IADocAccessorRecord
    with routine descriptors referring to routines that
    call through to the Mac OS file system.  It stores
    a record containing information about the file
    in the docAccessor field of the IADocAccessorRecord
    record.  If an error occurs, th function returns NULL.  */

static OSStatus NewIADocAccessorRec(
            FSSpec *targetFile,
            IADocAccessorPtr *docAccRec) {
    IADocAccessorPtr docAcc;
    MyDocRefPtr refptr;
    OSStatus err;

    refptr = NULL;

    refptr = (MyDocRefPtr) NewPtrClear(sizeof(MyDocumentReference));
    if (refptr == NULL) goto bail;
    refptr->spec = *targetFile;
    refptr->docOpen = false;
    refptr->refnum = 0;

    docAcc = (IADocAccessorPtr) NewPtrClear(sizeof(IADocAccessorRecord));
    if (docAcc == NULL) { err = memFullErr; goto bail; }
    docAcc->docAccessor = (IADocAccessorRef) refptr;

    docAcc->OpenDoc = NewIADocAccessorOpenProc(MyIADocAccessorOpenProc);
    if (docAcc->OpenDoc == NULL) { err = memFullErr; goto bail; }

    docAcc->CloseDoc = NewIADocAccessorCloseProc(MyIADocAccessorCloseProc);
    if (docAcc->CloseDoc == NULL) { err = memFullErr; goto bail; }

    docAcc->ReadDoc = NewIADocAccessorReadProc(MyIADocAccessorReadProc);
    if (docAcc->ReadDoc == NULL) { err = memFullErr; goto bail; }

    docAcc->SetReadPosition = NewIASetDocAccessorReadPositionProc(
        MyIASetDocAccessorReadPositionProc);
    if (docAcc->SetReadPosition == NULL) { err = memFullErr; goto bail; }

    docAcc->GetReadPosition = NewIAGetDocAccessorReadPositionProc(
        MyIAGetDocAccessorReadPositionProc);
    if (docAcc->GetReadPosition == NULL) { err = memFullErr; goto bail; }

    docAcc->GetEOF = NewIAGetDocAccessorEOFProc(
        MyIAGetDocAccessorEOFProc);
    if (docAcc->GetEOF == NULL) { err = memFullErr; goto bail; }

    *docAccRec = docAcc;

    return noErr;
```

```
bail:
    if (refptr != NULL) DisposePtr((Ptr) refptr);
    if (docAcc != NULL) {
        if (docAcc->OpenDoc != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) docAcc->OpenDoc);
        if (docAcc->CloseDoc != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) docAcc->CloseDoc);
        if (docAcc->ReadDoc != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) docAcc->ReadDoc);
        if (docAcc->SetReadPosition != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) docAcc->SetReadPosition);
        if (docAcc->GetReadPosition != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) docAcc->GetReadPosition);
        if (docAcc->GetEOF != NULL)
            DisposeRoutineDescriptor((UniversalProcPtr) docAcc->GetEOF);
        DisposePtr((Ptr)docAcc);
    }
    return err;
}

    /* DisposeIADocAccessorRec releases a IADocAccessorRecord
    allocated by NewIADocAccessorRec.  All o the sub
    fields are deallocated, and, if the file is open,
    it is closed before the structure is deallocated. */

static void DisposeIADocAccessorRec(IADocAccessorPtr docAcc) {
    MyDocRefPtr refptr;

        /* destroy the document reference */
    refptr = (MyDocRefPtr) docAcc->docAccessor;

        /* make sure the file is closed - incase we're aborting */
    if (refptr->docOpen) FSClose(refptr->refnum);

    DisposePtr((Ptr) refptr);

        /* release the accessor structure */
    DisposeRoutineDescriptor((UniversalProcPtr) docAcc->OpenDoc);
    DisposeRoutineDescriptor((UniversalProcPtr) docAcc->CloseDoc);
    DisposeRoutineDescriptor((UniversalProcPtr) docAcc->ReadDoc);
    DisposeRoutineDescriptor((UniversalProcPtr) docAcc->SetReadPosition);
    DisposeRoutineDescriptor((UniversalProcPtr) docAcc->GetReadPosition);
    DisposeRoutineDescriptor((UniversalProcPtr) docAcc->GetEOF);
    DisposePtr((Ptr) docAcc);
}
```

Listing 17. Routines for allocating and initializing a `IADocAccessorRecord`.

In Listing 17, we use File Manager calls to access the file. For tracking information used by the File Manager, we store a pointer to a private structure containing that information in the `docAccessor` field of the `IADocAccessorRecord`.

Back to top

Calling a Text Extractor Plug-in

The routine provided in Listing 18 calls the Text Extractor Plug-in to gather textual information from a file. The text gathered from the file is passed back to the caller through a routine the caller provides as a parameter.

```
    /* kETBufferSize determines the size of the buffer
    allocated for retrieving chunks of text. */

#define kETBufferSize (1024*1)

    /* TextSinkProc is a call back routine provided by the
    caller.  Text will be passed to this routine as it is
    extracted from the file. */


typedef OSErr (*TextSinkProc)(void* text, long length, long refcon);

    /* ExtractTextFromFile calls the Text Extractor Plug-in
    referred to by *theExtractor to extract text from the
    file referred to by *targetFile.  While extracting text,
    the text will be sent to the TextSinkProc provided by
    the textsink parameter. refcon is a value passed through
    to the TextSinkProc in its refcon parameter.  */

static OSErr ExtractTextFromFile(
            FSSpec *targetFile,
            FSSpec *theExtractor,
            TextSinkProc textsink, long refcon) {
    ExtractorRecPtr extractor;
    IAPluginInitBlockPtr initblock;
    IADocAccessorPtr accRec;
    UInt32 pluginVersion;
    Boolean exInited, docOpen;
    IADocRef docRef;
    Ptr etBuffer;
    UInt32 bytecount;
    OSStatus err;
    IAPluginRef pluginRef;

        /* set up locals to a known state */

    extractor = NULL;
    initblock = NULL;
    accRec = NULL;
    exInited = false;
    docOpen = false;
    etBuffer = NULL;

        /* initialize the plug-in */

    err = OpenExtractor(theExtractor, &extractor);
    if (err != noErr) goto bail;

        /* initialize the callbacks used by the
        plug-in for basic memory tasks.  */

    err = NewIAPluginInitBlock(&initblock);
    if (err != noErr) goto bail;

        /* call the plug-in's initialization routine.  */

    err = extractor->PluginInit(initblock, &pluginRef);
    if (err != noErr) goto bail;
    exInited = true;

        /* query the plug-in to find out if we're using
        the interface we're using is in sync with the
        interface it was built to use. */

    err = extractor->GetExtractorVersion(pluginRef, &pluginVersion);
    if (err != noErr) goto bail;
    if (pluginVersion != kIAExtractorVersion1)
        { err = errIAParamErr; goto bail; }

        /* initialize the callbacks used by the
        plug-in for file input with our document.  */

    err = NewIADocAccessorRec(targetFile, &accRec);
    if (err != noErr) goto bail;

        /* allocate a memory buffer for reading */

    etBuffer = NewPtr(kETBufferSize);
    if (etBuffer == NULL) { err = memFullErr; goto bail; }

        /* call the plug-in and ask it to open the document
        for input. */

    err = extractor->OpenDocument(pluginRef,  accRec, &docRef);
    if (err != noErr) goto bail;
    docOpen = true;

        /* Here, we loop until the plug-in returns no more bytes */

    while (true) {

            /* attempt to fill the entire buffer with text. */

        bytecount = kETBufferSize;
        err = extractor->GetNextTextRun(docRef, etBuffer, &bytecount);

            /* if some other error occurs, such as eofErr...
            we exit... */

        if (err != noErr) goto bail;
```

```
            /* errIAEndOfTextRun is returned when the language
            encoding changes.  in this case, we do nothing,
            but in some cases we may wish to do some additional
            processing. */

        if (err == errIAEndOfTextRun) {

            /* we don't check the bytecount
            here because conceivably errIAEndOfTextRun could
            be returned with a zero sized buffer simply to
            indicate the beginning of a new
            character encoding range in cases where the
            last call read all of the characters from the
            last encoding run. */

            /* normal termination occurs when zero bytes are
            returned. */

        } else if (bytecount == 0)
            break;


            /* at this point, we have a chunk of text from the
            from the document.  Here, we pass it back to the
            caller's sink. */

        err = textsink(etBuffer, bytecount, refcon);
        if (err != noErr) goto bail;
    }

        /* at this point, all of the text in the document
        has been read.  Now, we close down the document
        by asking the plug-in to close, disposing of the
        memory buffer, and then disposing the file input
        callback structure. DisposeIADocAccessorRec is
        defined in Listing 17. */

    extractor->CloseDocument(docRef);
    docOpen = false;
    DisposePtr(etBuffer);
    etBuffer = NULL;
    DisposeIADocAccessorRec(accRec);
    accRec = NULL;

        /* After closing the document, the plug-in
        is released.  This is done by calling the plug-in's
        termination procedure, releasing the memory allocation
        callbacks (DisposeIAPluginInitBlock is defined in
        Listing 16) and then releasing the plug-in's
        code fragment (CloseExtractor is defined in
        Listing 15). */

    extractor->PluginTerm(pluginRef);
    exInited = false;
    DisposeIAPluginInitBlock(initblock);
    initblock = NULL;
    CloseExtractor(extractor);
    extractor = NULL;

        /* return success */

    return noErr;

bail:

        /* error handling code.  note, ordering of the
        recovery statements is important. */

    if (docOpen) extractor->CloseDocument(docRef);
    if (etBuffer != NULL) DisposePtr(etBuffer);
    if (accRec != NULL) DisposeIADocAccessorRec(accRec);
    if (exInited) extractor->PluginTerm(pluginRef);
    if (initblock != NULL) DisposeIAPluginInitBlock(initblock);
    if (extractor != NULL) CloseExtractor(extractor);
    return err;
}
```

Listing 18. Sample routine for that calls a Text Extractor Plug-in.

The routine provided in Listing 18 performs the actual text extraction by calling the plug-in's routines directly. In this example, no attention is paid to the language encoding or character encoding, but this example could easily be modified to return this information. This routine uses structures and calls routines defined in Listing 15, Listing 16, and Listing 17.

Back to top

Index of Code Listings

The following code listings are provided in this document. Listings 5 through 14 define the content of the sample plug-in, and listings 15 through 18 illustrate how to call a plug-in from an application.

Back to top

References

Technote TN1141, "Extending and Controlling Sherlock."

Technote TN1180, "Sherlock's Find By Content Library."

RFC1521, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies." N. Borenstein, N. Freed. September 1993.

Back to top

Downloadables

Acrobat version of this Note (K). Download

Back to top

Technical Notes by API | Date | Number | Technology | Title
Developer Documentation | Technical Q&As | Development Kits | Sample Code