Technote 1141

Extending and Controlling

and the Find by Content Libraries

being a description of the Find facilities in Mac OS 8.5

By John Montbriand
Apple Worldwide Developer Technical Support

CONTENTS

Mac OS 8.5 includes several enhanced searching capabilities, known collectively as **Sherlock**. Previously, the Mac OS Find application allowed users to search mounted disk volumes for files based on information such as name, modification date, and file type. Sherlock retains this functionality, but also extends the user's search options to include both the content of files and the Internet.

Overview

To perform an Internet search, the Sherlock application sends query information to one or more Internet search sites. The information returned by the search sites is interpreted by the Sherlock application and then displayed for perusal. As each Internet search site has its own particular format for query and response information, the Sherlock application uses plug-ins that describe data formats expected and provided by individual Internet search sites for formatting queries and parsing response data. Internet search site providers interested in building their own Internet search site plug-ins will find directions for doing so in the Internet Search Plug-ins section.

AppleScript commands for accessing the new content-based search and Internet search facilities provided by the Sherlock application are available. These include commands for searching by content, a command for indexing volumes, and commands for performing Internet searches. These commands are discussed in greater detail in the AppleScript Support section.

The Sherlock application, when asked to open a file that was found by way of a content-oriented search, attaches information about the search and why the file was selected to the 'odoc' Apple event it passes to the Finder. The Finder passes this information along to applications as a property associated with the `'odoc'` Apple event. Applications can access this information and use it to perform further search and display actions when it is found in the `'odoc'` event. More information can be found in the kAEOpenDocuments section.

Find By Content is a new system-level facility implemented as a Code Fragment Manager library. The Sherlock application is a client of Find By Content and utilizes its search facilities for performing content-based searches. Developers interested in using the Find By Content services from within their applications may do so by linking against the PowerPC Code Fragment Manager library named "Find By Content" (without the quotes). Routine descriptions and examples are provided in the Find By Content section below.

Internet Search Plug-ins

The "Search Internet" feature in the Sherlock application allows users to perform Internet searches using one or more Internet search engines. The Sherlock application itself contains no information about the exact data formats expected or generated by individual Internet search engines; when accessing any particular Internet search site, the Sherlock application uses a search plug-in file that describes the data formats both expected by the site for queries and produced by the site in its responses to queries. Internet Search Interface Language (ISIL) is the language used in search plug-in files so that Internet search site administrators may provide their own search plug-in files.

ASCII text describing the search site is contained in a search plug-in's data fork. The resource fork may be used for custom icons, Finder strings, et cetera. Search plug-in files have the creator code `'fndf'` and the file type `'issp'` and will be only recognized by the Sherlock application when they reside in the "Internet Search Sites" Folder (FindFolder type = `'issf'`). When dropped onto the System Folder's (closed) icon, files of type `'issp'` are auto-routed to the "Internet Search Sites" folder.

ISIL is modeled closely after the HTML it is used to describe, so HTML authors familiar with the syntax should have little or no trouble creating their own search plug-in files. An exact specification of the language can be found in the Internet Search Interface Language BNF section, and the sections that follow discuss the language in greater detail.

To create a search plug-in file, you will need a text editor program -- Simple Text will do -- and a utility that will allow you to change the plug-in's file type. The basic steps for editing a search plug-in file are:

1. Open or create and then edit the file using your text editor program.
2. Save any changes you make and close the file.
3. Change the file type of the file from `'TEXT'` to `'issp'`.
4. Test your file (now a Sherlock plug-in) using the Sherlock application.
   If satisfied, you're done: stop.

5. Change the file type of the search plug-in from `'issp'` to `'TEXT'`.
6. Go to the first step in this list.

If your text editor edits any file regardless of type and does not change the types of the files it edits, you can skip steps 3 and 5.

The Sherlock application scans the "Internet Search Sites" only once when it is starting up. You should restart the Sherlock application each time you would like to test your search site file.

Search Plug-in Files

Search plug-in files contain ASCII text formatted similarly to the HTML text used to define web pages. Accordingly, terminology used to describe HTML is used in this document's description of ISIL syntax. Information describing an Internet search site is contained in a block labeled with the SEARCH tag. This block is used to describe how the Sherlock application sends queries to an Internet search site, and it includes information such as the site's URL, the HTTP command used to send a query, and query parameters. Listing 1 illustrates the typical layout for a SEARCH block.

Listing 1. Typical layout for a SEARCH block in a search plug-in file:

```
<SEARCH
    name = "<search engine name>"
    method = ["get" | "post"]
    action = "<url to address>"
    [update = "<url containing update file>"]
    [updateCheckDays = "<days between update pings>"]
    [description = "<human-readable-description">]
    [bannerImage = "<url containing banner image>"]
    [bannerLink = "<url to load when banner clicked>"]>

....

<INPUT
    name = "<input name>"
    value = "<value>"
    [mode = "results"]>
<INPUT
    name = "<input name>"
    value = "<value>"
    [mode = "browser"] >

....

<INPUT
    name = "<input name>"
    user>

....

<INTERPRET
    [bannerStart = "<text>"]
    [bannerEnd = "<text>"]

    [relevanceStart = "<text>"]
    [relevanceEnd = "<text>"]
```

```
    [resultListStart = "<text>"]
    [resultListEnd = "<text>"]
    [resultItemStart = "<text>"]
    [resultItemEnd = "<text>"]
    [skipLocal=true]

    [charset = "<text>"]
    [resultEncoding = <integer>]
    [resultTranslationEncoding = <integer>]
    [resultTranslationFont = "<text>"]>

....

</SEARCH>
```

Search blocks begin with the <SEARCH ....> tag (containing a number of attributes, as described in Table 1) and end with a </SEARCH> tag. Within a typical search block describing an Internet search site, there will be one or more INPUT tags and an INTERPRET tag. The SEARCH block attributes describe the search site, how it is to be accessed, and where updates to the search plug-in file can be found.

Table 1. SEARCH block attributes

Attribute Name	Description
`name`	This is a human-readable name for the search plug-in.
`method`	The `method` attribute specifies the type of HTTP command that should be used for communications with the HTTP server. Currently, either `"GET"` or `"POST"` can be specified as the communications method.
`action`	Specifies the full URL to access the search server. Any relative links in the result list will be localized using this URL.
`update`	This is an optional attribute specifying where the most recent version of the search plug-in file is kept. If provided, the Sherlock application will periodically check this URL for changes. If the file at this URL is found to be more recent than the one that is currently installed, the Sherlock application will prompt the user to download the new file and automatically install it. The file located at this URL should be in BinHex format (but not otherwise compressed or encoded).
`dateCheckDays`	This is an optional attribute specifying the number of days between times when the `update` URL is checked for more recent versions of the search plug-in file. If this attribute is not present, the default value of 30 days is used.
`description`	This is an optional attribute containing text describing the search engine, its capabilities, and the content type of the search results. This text may be used for display in user interface facilities.

`bannerImage`	This is an optional attribute specifying an URL for an image that will be displayed in the details pane when any result from a query using this search plug-in is selected. Note: the banner properties of the `INTERPRET` tag will override this setting when there is a conflict.
`bannerLink`	This is an optional attribute specifying an URL that will be loaded when the banner image is clicked. Note: the banner properties of the `INTERPRET` tag will override this setting when there is a conflict.

The `INPUT` tags are used to construct the data field used in the GET or POST command sent to the HTTP server. The data field is constructed using the HTTP syntax and the `method` field determines the method that is used to query the server. A search block may contain one or more `INPUT` tags, but only one of the `INPUT` tags can be a `USER INPUT` tag.

`INPUT` tags may specify an optional mode attribute. The Sherlock application will send two types of queries: one when it is retrieving results and another when it sends a query URL to a browser. `INPUT` tags specifying the "results" mode (the default) are used by the Sherlock application when it sends queries to search sites that will be displayed in the list of search results in the Sherlock application's window. `INPUT` tags specifying "browser" will be included in query URLs sent to browser applications for display. For example, the following two `INPUT` tags may be present in a search plug-in file:

```
<input name="sv" value="AP" mode = "results">
<input name="sv" value="IS" mode = "browser">
```

Here, &sv=AP will be sent to the server when the Sherlock application will be used to display the results, and &sv=IS will be sent to the server when a web browser will be used to display the results.

The `INTERPRET` tag describes the format of the information returned from search queries sent to the site. This information allows the Sherlock application to extract individual search results from a query and format them into a list. Table 2 describes the various attributes that may be specified for an `INTERPRET` tag. Each attribute specified in the `INTERPRET` tag specifies a text pattern occurring in the result page delimiting some specific part of the results. When available, the Sherlock application will use these text patterns to extract search result information from the result pages returned by Internet search sites and build lists of items for display.

Table 2. `INTERPRET` tag attributes

Attribute Name	Description
`resultListStart`	Specifies the text pattern present at the beginning of the list of search results in the result page returned by the server. If `resultListStart` is not specified, then the Sherlock application will assume the result list begins at the top of the result page.

`resultListEnd`	Specifies the text pattern present at the end of the list of search results in the result page returned by the server. If `resultListEnd` is not specified, then the Sherlock application will assume the result list ends at the bottom of the result page. The `resultListStart` and `resultListEnd` attributes are used to define text patterns delimiting the list of results.
`resultItemStart`	Specifies a text pattern present at the beginning of each individual item in the list of results. When the text specified is matched in the result page, only links immediately following the text pattern will be included in the list of results displayed for the user.
`resultItemEnd`	Specifies a text pattern present at the end of the text used to describe an item in the list of results. Text between a result's link and this text pattern will be presented in the details pane. The `resultItemStart` and `resultItemEnd` attributes are used to define text patterns delimiting individual items in the list of results returned by the server.
`bannerStart`	Specifies a text pattern used to locate the banner image to be displayed for the search results. The first link following the text pattern will be used as the `bannerLink` and the first image following the text pattern will be used as the `bannerImage`. If the `bannerStart` attribute is specified and the text pattern is matched, then the `bannerLink` and `bannerImage` will override those attributes specified in the `SEARCH` tag.
`bannerEnd`	Specifies a text pattern marking the end of the banner information. The search for a `bannerImage` and `bannerLink` will not proceed beyond this text pattern in the result page. The text patterns defined in the `bannerStart` and `bannerEnd` attributes are used to delimit the banner information that may be present in the result page. If banner information is found in the result page, then it will will be used instead of any banner information specified in the `SEARCH` tag; otherwise, if no banner information is found, then the default banner information specified in the `SEARCH` tag will be used.
`relevanceStart`	Specifies a text pattern marking the beginning of the relevance information provided for each item in the list of results. When present, the first numeric text found after the pattern will be interpreted as the relevance of the item. Note: the numbers used to represent relevance scores should be between 0 and 100.
`relevanceEnd`	Specifies a text pattern marking the end of the relevance information. The search for relevance information will not proceed beyond this text pattern. The text patterns defined in the `relevanceStart` and `relevanceEnd` attributes are used to delimit the relevance score for each individual search result. Note: the numbers used to represent relevance scores should be between 0 and 100.

skipLocal	`skipLocal` is a boolean attribute. If `skipLocal` is true, then the Sherlock application will ignore links that refer to the same host as specified in the `ACTION` attribute in the `SEARCH` tag.
`charset`	The expected encoding of the HTML results. This attribute may be set to any value appropriate for the `charset` HTML meta tag.
`resultEncoding`	The encoding that the HTML results are in. This may be any integer constant defined in `<TextCommon.h>`.
`resultTranslationEncoding`	The encoding that the HTML results should be translated to. This may be any integer constant defined in `<TextCommon.h>`.
`resultTranslationFont`	the preferred font for the translated text

The attributes `charset`, `resultEncoding`, `resultTranslationEncoding`, and `resultTranslationFont` are for interpreting results returned with different character encodings. If the result page contains the HTML meta tag "charset", then the Sherlock application will use the Text Encoding Converter to translate the document into a Macintosh encoding.

It is possible, though, that the Sherlock application will not be able to recognize a text encoding by name. For these cases, search plug-in creators can explicitly specify the character encoding that will be used in responses to queries by using the `resultEncoding` attribute. The value specified for the `resultEncoding` attribute can be any integer constant defined in the file `<TextCommon.h>`. Similarly, `resultTranslationEncoding` is used to specify the text encoding that the document should be translated into before processing continues. The value used for this attribute is also an integer constant from `<TextCommon.h>`.

For example, if a result page returned from a search site was encoded using the "euc-jp" character set (in `<TextCommon.h>` "euc-jp" is defined as `kTextEncodingEUC_JP` = 2336) and we would prefer that it be translated to Mac Japanese (defined as `kTextEncodingMacJapanese` = 1 in `<TextCommon.h>`) and displayed using the "Osaka" font, then the following character translation values would be specified:

```
<interpret
resultEncoding = 2336
resultTranslationEncoding = 1
resultTranslationFont = "Osaka"
>
```

`INTERPRET` tags are optional, and all of the attributes within an `INTERPRET` tag are optional as well. If a `SEARCH` block does not contain an `INTERPRET` tag, then every link found in the result page will be treated as a result and the Sherlock application will present the entire list to the user as the results of her query

An Example

In this hypothetical example, we assume the Internet search site that we are writing the search plug-in file for is located at the URL <http://clarus.apple.com>. (As of this writing, this site does not exist, although the following text is written as if the site does exist. If the site did exist, it would presumably enable visitors to search for information regarding Clarus the Dogcow. An explanation of how visitors other than dogcattle would make use of the search results is beyond the scope of this document and is left as an exercise for the reader.)

Step 1: Describe the site in the opening SEARCH tag.

Using your web browser, go to the search site and view the HTML source for the web page. Somewhere in the HTML, you should find a FORM tag as follows:

```
<form action="http://clarus.apple.com/Titles" method="get" name="Search">
```

Or, it is possible that the action may be specified as a local string as follows:

```
<form action="/Titles" method="get" name="Search">
```

If the action is specified as a local string, then prefix it with the address in the SEARCH tag's action attribute. Using the information found here, we can construct the opening SEARCH tag for the search block:

```
<search
    name="Clarus"
    description = "The Clarus Search Site"
    action="http://clarus.apple.com/Titles/"
    method=get>
```

From the HTML source, we were able to determine that the action is http://clarus.apple.com/Titles/ and the method appropriate for communicating with the site is get. The name of the site and the description are values we set ourselves.

Step 2: Define the INPUT tags.

There are two ways to determine what inputs are expected by an Internet search site. The first method is to manually perform a query and look at the URL that is sent to the server. The second is to pick through the HTML to discover the information.

The Query Method. Looking at the query information is the simplest method. For example, if we go to the search site in our web browser and type the query string "coffee" and start a search, then we may observe a URL that looks like this:

```
http://clarus.apple.com/Titles?qt=coffee&nh=10
```

From which, we can locate the inputs. The inputs come after the "?" and are separated by ampersand characters [&]. In this query, the inputs are as follows:

```
qt=coffee
nh=10
```

Using this information, we can construct the following two INPUT tags:

```
<input name="qt" user>
```

```
<input name="nh" value="10">
```

There may be some optional parameters available on a search site, so trying different options and queries may yield more useful information.

The HTML Method. If the inputs are not present in the URL then they must be determined by looking at the HTML source. Here, we look for the `INPUT` tags present in the search site's web page to determine what will be used to describe the inputs. For example, suppose the first few lines of the HTML for a search site were formatted as follows:

```
<form action="/Titles" method="get" name="Search">
<table width="100%" cellspacing=0 cellpadding=3 border=0>
<tr><td colspan=4>
Search</td>
<td align=center><a href="/Help?pg=Help.HTML"><b>Tips</b></a>
</td></tr>
<tr><td colspan=5>
<input type="text" name="qt" value="" size="25" MAXLENGTH=255>
</td></tr>
<INPUT TYPE=hidden NAME="nh" VALUE="10">
</table>
</form>
```

Between the `<form>` and `</form>` tags, there are the two inputs relevant to accessing this search engine:

```
<input type="text" name="qt" value="" size="25" MAXLENGTH=255>
<INPUT TYPE=hidden NAME="nh" VALUE="10">
```

Again, this information can be used to construct the following two `INPUT` tags:

```
<input name="qt" user>
<input name="nh" value="10">
```

Experimenting with these input parameters and writing different types of query URLs can provide useful information about their meaning and use. For instance, after writing several variations of the query URL, we discovered that `nh` specifies the number of hits that should be returned in a response to a query. Rather than 10 hits at a time, we would prefer to see 25 hits, so we change the inputs as follows:

```
<input name="qt" user>
<input name="nh" value="25">
```

Now that the inputs have been determined, there is enough information to put together a complete search plug-in file:

```
<search
    name="Clarus Test"
    description = "The Clarus Search Site"
    action="http://clarus.apple.com/Titles/"
    method=get>
<input name="qt" user>
<input name="nh" value="25">
</search>
```

However, in this form, although it will be possible for queries to be sent and results to be displayed, the lack of an `INTERPRET` tag means that the results may not be displayed correctly. To ensure that they are, an `INTERPRET` tag should be added.

Step 3: Describe the results in the `INTERPRET` tag.

Determining the text delimiters located in the responses returned by Internet search engines requires examination of the HTML source returned as the response to one or more queries. From this data, we can determine text patterns delimiting interesting parts of the response information. For example, suppose the following were returned as a response to a query:

Listing 2. A sample HTML response to a query:

```
<HTML>
<HEAD><TITLE>Sample Results</TITLE></HEAD>
<BODY>

<A HREF="http://www.apple.com">
<IMG SRC="http://www.apple.com/main/elements/apple.gif"
 ALT="Apple Computer"
</A>

<P>
<SMALL>90%</SMALL>
<A HREF="http://www.apple.com/hotnews/">Hot News</A> 
Apple Hot News - http://www.apple.com/hotnews
<BR><A HREF="http://www.apple.com">Apple Computer</A>
</P>
<P>
<SMALL>85%</SMALL>
<A HREF="http://www.apple.com/products/">Apple Products</A>
 
Apple - Products - http://www.apple.com/products
<BR><A HREF="http://www.apple.com">Apple Computer</A>
</P>
</BODY>
</HTML>
```

From this information, we can see that the banner section is delimited by the text patterns "<BODY>" and "<P>" as follows:

```
bannerStart="<BODY>"
bannerEnd="<P>"
```

The List of results are delimited by the text patterns "</A>" and "</BODY>":

```
resultListStart="</A>"
resultListEnd="</BODY>"
```

Each item in the list of results is bracketed by the text patterns "<P>" and "</P>":

```
resultItemStart="<P>"
resultItemEnd="</P>"
```

And, the relevance score for each item is bracketed by the text patterns "<SMALL>" and "</SMALL>":

```
relevanceStart="<SMALL>"
relevanceEnd="</SMALL>"
```

Putting this all together, the complete search plug-in file would have the following contents:

```
<search
    name="Clarus Test"
    description = "The Clarus Search Site"
    action="http://clarus.apple.com/Titles/"
    method=get>
<input name="qt" user>
<input name="nh" value="25">
<interpret
    bannerStart="<BODY>"
    bannerEnd="<P>"
    resultListStart="</A>"
    resultListEnd="</BODY>"
    resultItemStart="<P>"
    resultItemEnd="</P>"
    relevanceStart="<SMALL>"
    relevanceEnd="</SMALL>">
</search>
```

Internet Search and XML Search Results

It is possible that a search engine may provide a separate machine-readable interface such as Extensible Markup Language (XML).

Listing 3. A sample XML document:

```
<searchResponse>
 <advertisement>
  <a href="http://www.advertiser.com">
  <img src="ad.gif">
  </a>
 </advertisement>

 <searchResults>
  <resultItem>
   <b><relevance>67%</relevance></b>
   <link><a href="http://www.foo.com">Title</a></link><br/>
   <summary>Summary</summary>
  </resultItem>
 </searchResults>
</searchResponse>
```

At the time of this document's creation, the XML specification is still under development; however, using the current state of the standard, the Internet Search Interface can be easily configured to interpret XML result lists. For example, the `INTERPRET` tag shown below illustrates how a search plug-in could be set up to interpret the XML document shown in Listing 3.

```
<interpret
   bannerStart = "<advertisement>"
   bannerEnd = "</advertisement>"
   resultListStart = "<searchResults>"
   resultListEnd = "</searchResults>"
   resultItemStart = "<resultItem>"
   resultItemEnd = "</resultItem>"
   relevanceStart = "<relevance>"
   relevanceEnd = "</relevance>">
```

Tips for Search Site Administrators

Comment-style Delimiters

The Sherlock application uses information provided by search plug-in files to extract information from HTML results returned from Internet search sites. Specifically, information in search plug-in files is used to find delimiters in the response information for the banner information and the search results. The question of the Sherlock application being able to find and display results consistently depends entirely on the search site remaining in sync with the formats specified in the search plug-in file. When the formats specified in the search plug-in file are based on anecdotal properties found in one or two search results files as in the example above, this sort of desynchronization can occur quite easily whenever small formatting changes are made in the result pages generated by a search site.

To avoid this problem, it is suggested that search site administrators include comments delimiting the interesting parts of response pages. By doing so, search plug-in files can be built to use the comment text as delimiters, and HTML formatting information included in result pages can be modified without risk of invalidating search plug-in files that have been built to access the search site. For example, the INTERPRET tags given below could be used to interpret the HTML response information shown in listing 4.

```
bannerStart="<!-- BANNER START -->"
bannerEnd="<!-- BANNER END -->"
resultListStart="<!-- RESULT LIST START -->"
resultListEnd="<!-- RESULT LIST END -->"
resultItemStart="<!-- RESULT ITEM START -->"
resultItemEnd="<!-- RESULT ITEM END -->"
relevanceStart="<!-- RELEVANCE START -->"
relevanceEnd="<!-- RELEVANCE END -->"
```

Using these text delimiters, the search provider can freely add additional formatting information to their response pages without being concerned about invalidating any search plug-in files currently in use. This approach is strongly recommended for all search site providers creating search plug-in files.

Listing 4. A simple HTML response to a query that includes delimiting comments:

```
<HTML>
<HEAD><TITLE>Sample Results</TITLE></HEAD>
<BODY>

<!-- BANNER START -->
<A HREF="http://www.apple.com">
<IMG SRC="http://www.apple.com/main/elements/apple.gif"
 ALT="Apple Computer"
</A>
<!-- BANNER END -->

<!-- RESULT LIST START -->

<!-- RESULT ITEM START -->
<P>
<SMALL>
<!-- RELEVANCE START -->
90%
<!-- RELEVANCE END -->
</SMALL>
<A HREF="http://www.apple.com/hotnews/">Hot News</A> 
Apple Hot News - http://www.apple.com/hotnews
<BR><A HREF="http://www.apple.com">Apple Computer</A>
</P>
<!-- RESULT ITEM END -->

<!-- RESULT ITEM START -->
<P>
<SMALL>
<!-- RELEVANCE START -->
85%
<!-- RELEVANCE END -->
</SMALL>
<A HREF="http://www.apple.com/products/">Apple Products</A>
 
Apple - Products - http://www.apple.com/products
<BR><A HREF="http://www.apple.com">Apple Computer</A>
</P>
<!-- RESULT ITEM END -->

<!-- RESULT LIST END -->

</BODY>
</HTML>
```

Banner Advertisements

The Sherlock application uses the first HTML anchor (that includes a hypertext jump and an image) found in the banner section as the banner image. For best results, banner advertisements should be enclosed in an HTML anchor that includes both an hypertext jump (HREF attribute) and an IMG tag that includes a SRC attribute and, preferably, an ALT attribute. For example, the HTML anchor shown below illustrates the suggested format for banner advertisements:

```
<A HREF="http://www.apple.com">
<IMG SRC="http://www.apple.com/main/elements/apple.gif"
ALT="Apple Computer"
</A>
```

Result Lists

When interpreting search results, the Sherlock application identifies results by looking for HTML anchors containing hypertext jump attributes. At least one anchor including an hypertext jump (HREF attribute) should occur between the text patterns specified in `resultItemStart` and `resultItemEnd` or `resultItemStart`. The Sherlock application will attempt to interpret HTML results between these text patterns and expects to find at least one such anchor.

Internet Search Interface Language BNF

All tags are case-insensitive and white space is ignored.

```
<search-interface>   ::= <search-start> <input-interp-list> <search-end>

<search-start>       ::= "<search " (<search-attribute> <req-S>)* ">"
<search-attribute>   ::= <name> | <method> | <action> | <update> |
                         <updateCheckDays> | <description> |
                         <banner-link> | <banner-image>
<name>               ::= "name" <attrib-assign>
<method>             ::= "method" <attrib-assign>
<action>             ::= "action" <attrib-assign>
<update>             ::= "update" <attrib-assign>
<updateCheckDays>    ::= "updateCheckDays" <attrib-assign>
<description>        ::= "description" <attrib-assign>
<banner-link>        ::= "bannerlink" <attrib-assign>
<banner-image>       ::= "bannerimage" <attrib-assign>


<input-interp-list>  ::= <input>* <interpret>? <input>*
<input>              ::= "<input " (<input-attribute> <req-S>)* ">"
<input-attribute>    ::= <name> | <value> | <user-select>
<value>              ::= "value" <attrib-assign>
<user-select>        ::= "user"

<interpret>          ::= <interpret " (<interpret-attribute> <req-S>)* ">"
<interpret-attribute>::= <rl-start> | <rl-end> | <ri-start> | <ri-end>
                         <banner-start> | <banner-end> | <rel-start> |
                         <rel-end> | <skip-local>
<rl-start>           ::= "resultListStart" <attrib-assign>
<rl-end>             ::= "resultListEnd" <attrib-assign>
<ri-start>           ::= "resultItemStart" <attrib-assign>
<ri-end>             ::= "resultItemEnd" <attrib-assign>
<banner-start>       ::= "bannerStart" <attrib-assign>
<banner-end>         ::= "bannerEnd" <attrib-assign>
<rel-start>          ::= "relevanceStart" <attrib-assign>
<rel-end>            ::= "relevanceEnd" <attrib-assign>
<skip-local>         ::= "skipLocal"
```

```
<attrib-assign>      ::= <opt-S> "=" <opt-S> <attrib>
<attrib>             ::= <quotestr> | <doublequotestr> | <noquotestr>
<quotestr>           ::= '\'' [^']* '\''
<doublequotestr>     ::= '"' [^"]* '"'
<noquotestr>         ::= [^ ]*
<req-S>              ::= (#x20 | #x09 | #x0D | #x0A)+
<opt-S>              ::= (#x20 | #x09 | #x0D | #x0A)*
```

AppleScript Support

The new search facilities provided by the Sherlock application can be accessed from AppleScript scripts. AppleScript scripts can ask the Sherlock application to perform an Internet search using one or more Internet Search Sites or search for files with specific content on local or remote volumes. Each of these commands returns the results of the search as a string that can be used elsewhere in your script. Optionally, AppleScript scripts can ask the Sherlock application to display the results of the search.

Searching the Internet

Internet based searches use the "search Internet" command. The "search Internet" command allows AppleScript scripts to specify the Internet search sites that will be used in the search along with query information. The query information can be provided as either a string or as a reference to a file containing the query information (but not both). Results of the search are returned as a string, and it is possible to specify that the Sherlock application display the results. Definition 1 includes the "search Internet" entry from the Sherlock application's AppleScript dictionary.

Definition 1. The "search Internet" dictionary entry from the Sherlock application

search Internet: Search the Internet

search Internet string -- the Internet sites to search, optional

[**for** string] -- the text to look for...
[**using** alias] -- ...or a saved Find file containing the query
[**display** boolean] -- Specifies whether or not to display the result (default is without display)

Result: string -- the URLs that match the query

It is important to remember that the "for" and "using" parameters are mutually exclusive and cannot be used together in one command. Either the query information is provided as a string or it is provided in a file. If the display parameter is true, then the Sherlock application will display the results of the search.

The "using" parameter allows query information stored in a file to be used rather than a query string. To create such a file, use the "Save Search Criteria" command in the Sherlock application's File menu.

The direct object to this command is a list of Internet search site names. If the list of Internet search site names is not specified and the "for string" parameter is used, then the same sites that were used in the last Internet search will be used in the search. The list of Internet sites is ignored when the "using alias" parameter is specified.

Searching Files

Two AppleScript commands are provided for access to the Find by Content facilities in the Sherlock application. The first command allows AppleScript scripts to perform searches based on contents of files and the second allows AppleScript scripts to create or update index files on particular volumes that are used by Find By Content. The AppleScript dictionary entry for the "search" command is shown in Definition 2 and the "index volumes" command is shown in Definition 3. The "search" command allows AppleScript scripts to perform searches based on file contents.

Definition 2. The "search" dictionary entry from the Sherlock application

> **search**: Search disks or servers
>
> **search** alias -- the volumes or folders to search, optional
>
> [**for** string] -- the text to look for...
> [**similar to** alias] -- ...or file(s) containing text for Find by Content...
> [**using** alias] -- ...or a saved Find file containing the query
> [**display** boolean] -- (default is without display) Specifies whether or not to display the result
>
> Result: alias -- the files that match the query

In the "search" command, the parameters "for", "similar to", and "using" are mutually exclusive parameters and may not be used together in the same command.

As in the Internet search command, the "using" parameter allows query information stored in a file to be used rather than a query string. To create such a file, use the "Save Search Criteria" command in the Sherlock application's File menu.

The direct object to the "search" command is a list of volumes or folders to search. If no list of volumes is provided and either the "search for" or the "search similar to" parameter is used, then the "search" command will search all local, indexed volumes. When the "using" parameter is specified, the list of volumes is ignored.

Indexing Volumes

Before the Find By Content facilities can be used to search a volume, the volume must contain an index. Index files are stored in an invisible folder called "TheFindByContentFolder" located in a volume's root directory and they contain necessary information for performing content-based searches. A volume cannot be searched by the Find By Content facilities unless it contains an index. AppleScript scripts can ask the Sherlock application to either update or create an index file for one or more volumes.

Definition 3. The "index volumes" dictionary entry from the Sherlock application.

> **index volumes**: Create or update the index(es) of the specified volume(s)
>
> **index volumes** alias -- list of volumes

The Optional `kAEOpenDocuments` Apple Event Parameter

To provide applications with information useful in selecting and displaying parts of documents that users may be interested in, when the user opens a file that was located by way of a content-based search from within one of the Sherlock application's windows, the Sherlock application will insert information about the search that led to the file into the `kAEOpenDocuments` (`'odoc'`) Apple event that is used to open the file. The Sherlock application opens files by sending `kAEOpenDocuments` Apple events to the Finder. The Finder, when receiving the `kAEOpenDocuments` Apple event, launches the application owning the document and passes the event to the application.

This type of `kAEOpenDocuments` Apple event contains an additional `keyAEPropData` (defined in `AERegistry.h`) parameter. Among the properties in the `keyAEPropData` parameter there is one identified using the keyword `'srwd'` that contains the original query string used to locate the file. The `'srwd'` property's data is formatted as a C-style string.

Listing 5. Retrieving the search words from and `'odoc'` Apple event:

```
OSErr GetSearchWordsFromAppleEvent(AppleEvent* inAppleEvent,
                                   char* theText, long *maxLength)
{
    OSErr err;
    AERecord propData;
    DescType outType;

        /* set up our variables */
    AECreateDesc(typeNull, NULL, 0, &propData);
    if (maxLength == NULL || theText  == NULL) return paramErr;
    if (*maxLength < 255) return paramErr;

        /* get the property data from the Apple event */
    err = AEGetParamDesc(inAppleEvent,
                  keyAEPropData, typeAERecord, &propData);

        /* extract the search words information */
    if (err == noErr)
        err = AEGetKeyPtr(&propData, 'srwd', typeChar,
                  &outType, theText, *maxLength, maxLength);

        /* clean up and return */
    AEDisposeDesc(&propData);
    return err;
}
```

The Example shown in Listing 5 illustrates how an application may extract the query information from an `kAEOpenDocuments` Apple event. Here, the routine attempts to retrieve the `keyAEPropData` parameter and then it attempts to extract the `'srwd'` information from the property data. If no problems occur and the `'srwd'` data is present, then the original query text will be returned in the buffer pointed to by theText, `*maxLength` will be set to the length of the string (including the trailing zero byte), and the function will return `noErr`.

The presence of this additional parameter will not affect the behavior of existing applications built according to the guidelines set forth in the "Responding to Apple Events" chapter of *Inside Macintosh* : Interapplication Communication. However, developers may choose to take advantage of this new information when it is present in an Apple event as a clue pointing to the part of the document that the user would like to see first. (The presence of the `'srwd'` information in an `kAEOpenDocuments` Apple event implies that the user conducted a search by content and then selected and opened the document from within the list of files that were found in the search.) For example, an application may choose to highlight all occurrences of the words in the string, view the first occurrence of a word from the string, or open its find window with one or more of the query terms.

In some cases, however, it is possible that some or all of the words in the query string may not appear in the document being opened. In a normal search based on a query phrase, Find By Content will locate files that contain one or more of the words in the query. But, when a user selects one or more documents found in a previous search and requests "similar" documents, then it is possible that some of the documents found may not contain any of the words from the query string specified in the original search. Developers accessing the `'srwd'` property should plan for the possibility that some or all of the keys in the query string may not be present in the document being opened.

Find By Content

The Find By Content (FBC) facilities provided in Mac OS 8.5 are implemented in a PowerPC Code Fragment Manager library that resides in the "Extensions" folder. The Sherlock application is a client of FBC, accessing FBC services through this shared library. Developer applications can also access the search facilities provided by this library. This section describes how developers can create products that access the new FBC facilities through this shared library.

Compiler interfaces to FBC are found in the C header file `<FindByContent.h>`. And, for linking purposes, the Code Fragment Manager library implementing FBC is named "Find By Content" (without the quotes). Developers using the FBC routines described herein should weak-link against this library, and then check the Gestalt selectors from within their application before calling any of these routines.

Determining if Find By Content is Available

FBC defines two `Gestalt` selectors. Clients of FBC must verify that correct version of the implementation is available before making any of these calls, and will want to check the FBC indexing state before performing any searches.

```
enum
{
    gestaltFBCVersion           = 'fbcv',
    gestaltFBCCurrentVersion    = 0x0011
};
```

The `gestaltFBCVersion` selector returns the version of FBC that is installed on the computer. Developers can compare this version with the version of the interface with which they have compiled their programs using the `gestaltFBCCurrentVersion` to determine if it is safe to make any calls to FBC. If `gestaltFBCVersion` produces some version other than the version of the interface your application has been compiled to run with, then your application should not make any calls to FBC.

```
enum
{
    gestaltFBCIndexingState     = 'fbci',
    gestaltFBCindexingSafe      = 0,
    gestaltFBCindexingCritical  = 1
};
```

The `gestaltFBCIndexingState` selector returns information about the current indexing status of FBC. At any given time, the indexing status will be either `gestaltFBCindexingSafe` or `gestaltFBCindexingCritical`. If the status is `gestaltFBCindexingCritical`, then any search will result in a synchronous wait until the state returns to `gestaltFBCindexingSafe`. When the FBC indexing state returned is `gestaltFBCindexingSafe`, then all searches will execute immediately. To avoid synchronous waits, developers should check the `gestaltFBCIndexingState` selector and only make calls to FBC when the indexing state returned is `gestaltFBCindexingSafe`.

Working with Search Sessions

FBC allows client applications to open and close a "search session". A search session contains all of the information about a search, including the list of matched files after the search is complete. Clients of FBC can obtain references to search sessions, modify them, and query their state using the routines defined in this section. References to search sessions are defined as an opaque pointer type owned by the FBC library.

```
typedef struct OpaqueFBCSearchSession* FBCSearchSession;
```

Developers should only access the search session structure using the routines defined herein. This includes using the appropriate FBC routines for duplicating and disposing of search sessions. Search sessions are complex memory structures that contain pointers to other data that may need to be copied when a search session is duplicated or disposed of when a search session is deallocated.

The normal sequence of actions one takes when using the FBC library is to create a search session, configure the search session to target specific volumes, perform the search, query the search results, and dispose of the search. Other possibilities for searches include the ability to reinitialize a search session and use it over again for another search, to provide backtracking by cloning search sessions and performing additional searches using the clones, or to limit search results to files found in particular directories.

Setting up a Search Session

Creating a new session and preparing it for a search, as shown in Listing 6, requires at least two calls to the FBC library. In this example, a new search session is created and it is configured to search all local volumes that contain index files. The call to `FBCAddAllVolumesToSession` automatically configures the search session to search all indexed volumes.

Listing 6. Setting up a search session to search all local, indexed volumes:

```
/* SimpleSetUpSession allocates a new search session and
    returns a FBCSearchSession value in the *session
    parameter. if an error occurs, *session is left
    untouched. */

OSErr SimpleSetUpSession(FBCSearchSession* session)
{
    OSErr err;
    FBCSearchSession newsession;

        /* set up our local variables */
    err = noErr;
    newsession = NULL;
    if (session == NULL) return paramErr;

        /* create the new session */
    err = FBCCreateSearchSession(&newsession);
    if (err != noErr) goto bail;

        /* search all available local volumes */
    err = FBCAddAllVolumesToSession(newsession, false);
    if (err != noErr) goto bail;

        /* store our result and leave */
    *session = newsession;
    return noErr;

bail:
    if (newsession != NULL)
        FBCDestroySearchSession(newsession);
    return err;
}
```

FBC provides a complete set of routines for developers wanting more control over what volumes will be searched by the search session. Listing 7 illustrates how a new search session could be configured to search a particular set of volumes.

Listing 7. Setting up a session to search a particular set of volumes:

```
/* SetUpVolumeSession allocates a new search session and
    returns a FBCSearchSession value in the *session parameter.
    if vCount is not zero, then vRefNums points to an array of
    volume reference numbers for volumes that are to be searched.
    if any of the vRefNums refer to a volume without an index,
    paramErr is returned.  */

OSErr SetUpVolumeSession (FBCSearchSession* session,
                            UInt16 vCount, SInt16 *vRefNums)
{
    OSErr err;
    UInt16 i;
    FBCSearchSession newsession;

        /* set up our local variables */
    err = noErr;
    newsession = NULL;
    if (vCount == 0) return paramErr;
    if (session == NULL) return paramErr;
    if (vRefNums == NULL) return paramErr;

        /* create the new session */
    err = FBCCreateSearchSession(&newsession);
    if (err != noErr) goto bail;

        /* search the volumes specified in vRefNums */

    for (i=0; i<vCount; i++) {
        if (!FBCVolumeIsIndexed(vRefNums[i])) {
            err = paramErr;
            goto bail;
        } else {
            err = FBCAddVolumeToSession(newsession,
                                        vRefNums[i]);
            if (err != noErr) goto bail;
        }
    }

        /* store our result and leave */
    *session = newsession;
    return noErr;

bail:
    if (newsession != NULL)
        FBCDestroySearchSession(newsession);
    return err;
}
```

In this example, the `FBCAddVolumeToSession` routine is used to add volumes to the search session. Other routines for querying what volumes are currently targeted by a search session and removing volumes from that list are provided.

Once a search session has been configured to search a number of volumes, it can be used again after a search has been conducted without having to reconfigure its target volumes. After performing a search and examining the results, the search session can be prepared for another search by calling the routine `FBCReleaseSessionHits`. This routine releases all of the search results from the last search while leaving the list of target volumes intact.

Making a copy of a search session using the routine `FBCCloneSearchSession` will copy the list of target volumes to the duplicate search session.

Performing Searches

When FBC performs a search, it will generate a list of files that were matched. This list is referred to as the "hits", and it is stored inside of the search session. FBC can be asked to perform a content-based search using a query string containing a list of words, a similarity search based on one or more hits obtained in a previous search, or a similarity search based on a list of example files. Listing 8 illustrates how a query-based search can be performed. Here, the query is used to search for matching files on all local indexed volumes.

Listing 8. A Query based search of all local, indexed volumes:

```
OSErr SimpleFindByQuery (char *query, FBCSearchSession *session)
{
    OSErr err;
    FBCSearchSession newsession;

        /* set up locals, check parameters... */
    if (query[0] == 0) return paramErr;
    if (session == NULL) return paramErr;
    newsession = NULL;

        /* allocate a new search session */
    err = SimpleSetUpSession(&newsession);
    if (err != noErr) goto bail;

        /* Here is the call that does the actual search,
        storing the results in the search session. */
    err = FBCDoQuerySearch(newsession, query,
                                NULL, 0, 100, 100);
    if (err != noErr) goto bail;

        /* save the results and return */
    *session = newsession;
    return noErr;

bail:
    if (newsession != NULL)
        FBCDestroySearchSession(newsession);
    return err;
}
```

Searches conducted using either the routine `FBCDoExampleSearch` or the routine `FBCBlindExampleSearch` can be used to locate files that are similar to other files. Similarity searches will locate files with similar content to the files specified as examples. Examples can be specified as indexes referring to hits obtained from previous searches, or as a list of `FSSpec` records referring to files on disk.

All three of the search routines-- `FBCDoExampleSearch`, `FBCBlindExampleSearch`, and `FBCDoQuerySearch`--provide support for limiting the search results to files residing in one or more directories. To do this, clients provide a list of `FSSpec` records referring to target directories. The example in Listing 9 illustrates how to limit the results of a search to a particular set of directories.

Listing 9. Searching a particular set of directories:

```
enum {
    kMaxVols = 20,
    maxHits = 10,
    maxHitTerms = 10
};

OSErr RestrictedFindByQuery (char *query, UInt16 dirCount,
                                  FSSpec* dirList,
                                      FBCSearchSession* session)
{
    UInt16 vCount, i;
    SInt16 vRefNums[kMaxVols], normalVol;
    FBCSearchSession newsession;

    vCount = 0;
    newsession = NULL;
    if (dirList == NULL || dirCount == 0) return paramErr;
    if (query == NULL) return paramErr;
    if (*query == 0) return paramErr;
    if (session == NULL) return paramErr;

        /* collect all of the unique volume reference numbers
        from the list of FSSpecs provided in the parameters. */
    for (i=0; i<dirCount; i++) {
        Boolean found;
        HParamBlockRec pb;

            /* ensure the vRefNum is a volume
            reference number */
        pb.volumeParam.ioVRefNum = dirList[i].vRefNum;
        pb.volumeParam.ioNamePtr = NULL;
        pb.volumeParam.ioVolIndex = 0;
        if ((err = PBHGetVInfoSync(&pb)) != noErr) goto bail;
        normalVol = pb.volumeParam.ioVRefNum;

            /* make sure it's not already in the list */
        for (found = false, j=0; j<vCount; j++)
            if (vRefNums[j] == normalVol) {
                found = true;
                break;
            }
```

```
            /* add the volume to the list */
        if (!found && vCount < kMaxVols)
            vRefNums[vCount++] = normalVol;
    }

        /* set up a session to use the volumes we found */
    err = SetUpVolumeSession(&newsession, vCount, vRefNums);
    if (err != noErr) goto bail;

        /* Here is the call that does the actual search,
        storing the results in the search session. */
    err = FBCDoQuerySearch(newsession, (char*)queryTxt,
                    dirList, dirCount, maxHits, maxHitTerms);
    if (err != noErr) goto bail;

        /* save the result and return */
    *session = newsession;
    return noErr;

bail:
    if (newsession != NULL)
        FBCDestroySearchSession(newsession);
    return err;
}
```

Here, volume reference numbers extracted from the array of `FSSpec` records referring to target directories provided as a parameter are used to configure the volumes that will be searched by the search session. Then, the list of target directories is passed to the `FBCDoQuerySearch`.

Retrieving Information from a Search Session

After a search is conducted using a search session, the search session may contain information about one or more matching files. Clients can access information about individual hits including the file's `FSSpec` record, the words that were matched in the file, the "score" assigned to the file during the last search operation, and additional information about the file. Listing 10 illustrates how one could obtain information about each hit returned by a search.

Listing 10. Enumerating all of the files found in a search session:

```
   typedef OSErr (*HitProc) (FSSpec theDoc,
                             float score,
                             UInt32 nTerms,
                             FBCWordList hitTerms);

/* SampleHandleHits can be called after a search to enumerate
    the search results.  For each search hit, the hitFileProc
    function parameter is called with information describing
    the target.  */
OSErr SampleHandleHits (FBCSearchSession session,
                                    HitProc hitFileProc)
{
    OSErr err;
```

```
    UInt32 hitCount, i;
    FSSpec targetDoc;
    float targetScore;
    FBCWordList targetTerms;
    UInt32 numTerms;

        /* set up locals, check parameters */
    targetTerms = NULL;
    if (hitFileProc == NULL) return paramErr;
    if (session == NULL) return paramErr;

        /* count the number of hits in this session */
    err = FBCGetHitCount(session, &hitCount);
    if (err != noErr) goto bail;

        /* iterate through the hits */
    for (i = 0; i < hitCount; i++) {

            /* get the target document's FSSpec */
        err = FBCGetHitDocument(session, i, &targetDoc);
        if (err != noErr) goto bail;

            /* get the score for this document */
        err = FBCGetHitScore(session, i, &targetScore);
        if (err != noErr) goto bail;

            /* get a list of the words matched in
            this document */
        numTerms = maxHitTerms;
        err = FBCGetMatchedWords(session, i, &numTerms,
                                             &targetTerms);
        if (err != noErr) goto bail;

            /* call the call back routine provided as a
            parameter to do something with the information. */
        err = hitFileProc(&targetDoc, score, numTerms,
                                             targetTerms);
        if (err != noErr) goto bail;

            /* clean up before moving to the next iteration. */
        FBCDestroyWordList(targetTerms, numTerms);
        targetTerms = NULL;

    }

    return noErr;

bail:
    if (targetTerms != NULL)
        FBCDestroyWordList(targetTerms, numTerms);
    return err;
}
```

Find By Content Reference

This section provides a description of the CFM-based interfaces to the PowerPC FBC library. PowerPC applications using these routines link against the library named "Find By Content" (without the quotes).

Data Types

FBC provides the following data types. Storage management for these types is provided by the FBC library. Clients should not attempt to allocate or deallocate these structures using calls to the Memory Manager.

`FBCSearchSession`

```
typedef struct OpaqueFBCSearchSession* FBCSearchSession;
```

Search sessions created by FBC are referenced through pointer variables of this type. The internal format of the data referred to by this pointer is internal to the FBC library. Clients should not attempt to access or modify this data directly.

`FBCWordItem`

```
typedef char* FBCWordItem;
```

An ordinary C string. This type is used when retrieving information about hits from a search session.

`FBCWordList`

```
typedef FBCWordItem* FBCWordList;
```

An array of `WordItems`. This type is used when retrieving information about hits from a search session.

Allocation and Initialization of Search Sessions

The following routines can be used to allocate and dispose of search sessions. Storage occupied by search sessions is owned by the FBC library, and these are the only routines that should be used to allocate, copy, and dispose of search sessions.

FBCCreateSearchSession

```
OSErr FBCCreateSearchSession(
            FBCSearchSession* searchSession);
```

`searchSession` points to a variable of type `FBCSearchSession`.

`FBCCreateSearchSession` allocates a new search session and returns a reference to it in the variable pointed to by `searchSession`.

FBCDestroySearchSession

```
OSErr FBCDestroySearchSession(
            FBCSearchSession theSession);
```

`theSession` is a pointer to a search session.

`FBCDestroySearchSession` reclaims the storage occupied by a search session. This will include any volume configuration information and hits associated with the search session.

FBCCloneSearchSession

```
OSErr FBCCloneSearchSession(
            FBCSearchSession original,
            FBCSearchSession* clone);
```

`original` is a pointer to a search session.

`clone` points to a variable of type `FBCSearchSession`.

`FBCCloneSearchSession` creates a new search session and stores a pointer to it in the variable pointed to by the clone parameter. Information from the original search session that is copied to the new session includes the volumes targeted by the search session and all of the hits that may have been found in previous searches.

Configuring Search Sessions

Search sessions can be configured to limit searches to a particular set of volumes. These routines allow clients access to the set of volumes that will be searched by FBC.

FBCAddAllVolumesToSession

```
OSErr FBCAddAllVolumesToSession(
            FBCSearchSession theSession,
            Boolean includeRemote);
```

`theSession` is a pointer to a search session.

`includeRemote` is a `Boolean` value.

`FBCAddAllVolumesToSession` configures a search session to search all mounted volumes that have been indexed. If `includeRemote` is true, then remote volumes will be included in the search session's list of target volumes. Volumes that are not indexed are not added to search session's list of target volumes.

FBCSetSessionVolumes

```
OSErr FBCSetSessionVolumes(
            FBCSearchSession theSession,
            const SInt16 vRefNums[],
            UInt16 numVolumes);
```

`theSession` is a pointer to a search session.

`vRefNums` is an array of volume reference numbers (16-bit integers).

`numVolumes` is an integer value containing the number of volume reference numbers in the array `vRefNums`.

`FBCSetSessionVolumes` allows clients to add several volumes to the list of volumes targeted by a search session in a single call.

FBCAddVolumeToSession

```
OSErr FBCAddVolumeToSession(
            FBCSearchSession theSession,
            SInt16 vRefNum);
```

`theSession` is a pointer to a search session.

`vRefNum` is a volume reference number.

`FBCAddVolumeToSession` adds a volume to the list of volumes that will be searched by the search session. If the volume is not indexed, it will not be added to the list.

FBCRemoveVolumeFromSession

```
OSErr FBCRemoveVolumeFromSession(
            FBCSearchSession theSession,
            SInt16 vRefNum);
```

`theSession` is a pointer to a search session.

vRefNum is a volume reference number.

FBCRemoveVolumeFromSession removes the specified volume from the list of volumes that will be searched by the search session.

FBCGetSessionVolumeCount

```
OSErr FBCGetSessionVolumeCount(
            FBCSearchSession theSession,
            UInt16* count);
```

theSession is a pointer to a search session.

count is a pointer to a 16-bit integer where the result is to be stored.

FBCGetSessionVolumeCount returns, in *count, the number of volumes in the list of volumes that will be searched by the search session.

FBCGetSessionVolumes

```
OSErr FBCGetSessionVolumes(
            FBCSearchSession theSession,
            SInt16 vRefNums[],
            UInt16* numVolumes);
```

theSession is a pointer to a search session.

vRefNums is an array of volume reference numbers (16-bit integers).

*numVolumes is a pointer to a 16-bit integer. On input, this will be the number of elements that can be stored in vRefNums, and on output it will be the number of elements actually stored in vRefNums.

FBCGetSessionVolumes returns the list of volumes that will be searched by the search session in the array pointed to by vRefNums. *numVolumes is set to the number of volume reference numbers returned in the array.

Executing a Search

FBC provides three different routines for conducting searches that are described in this section.

FBCDoQuerySearch

```
OSErr FBCDoQuerySearch(
            FBCSearchSession theSession,
            char* queryText,
            const FSSpec targetDirs[],
            UInt32 numTargets,
            UInt32 maxHits,
            UInt32 maxHitWords);
```

`theSession` is a pointer to a search session.

`queryText` refers to a C-style string containing the search terms.

`targetDirs` points to an array of `FSSpec` records that refer to directories. If `numTargets` is zero, then this parameter can be set to `NULL`.

`numTargets` contains the number `FSSpec` records in the array pointed to by `targetDirs`.

`maxHits` the maximum number of hits that should be returned.

`maxHitWords` the maximum number of hit words that will be reported.

`FBCDoQuerySearch` performs a search based on the search terms found in `queryText`. If the `targetDirs` parameter is present (`numTargets` is not zero), then only files residing in the directories specified in `targetDirs` will be included in the hits found by the search.

`FBCDoExampleSearch`

```
OSErr FBCDoExampleSearch(
            FBCSearchSession theSession,
            const UInt32* exampleHitNums,
            UInt32 numExamples,
            const FSSpec targetDirs[],
            UInt32 numTargets,
            UInt32 maxHits,
            UInt32 maxHitWords);
```

`theSession` contains a pointer to a search session. This session must contain a hit list generated by a previous search.

`exampleHitNums` points to an array of 32 bit integers.

`numExamples` contains the number of integers in the array pointed to by `exampleHitNums`.

`targetDirs` points to an array of `FSSpec` records that refer to directories. If `numTargets` is zero, then this parameter can be set to `NULL`.

`numTargets` contains the number `FSSpec` records in the array pointed to by `targetDirs`.

`maxHits` the maximum number of hits that should be returned.

`maxHitWords` the maximum number of hit words that will be reported.

`FBCDoExampleSearch` performs an example-based or "similarity" search using hits found in a previous search as examples. `exampleHitNums` points to an array of long integers containing the indexes of one or more of the hits that are to be used as example files. If the `targetDirs` parameter is present (`numTargets` is not zero), then only files residing in the directories specified in `targetDirs` will be included in the hits found by the search.

FBCBlindExampleSearch

```
OSErr FBCBlindExampleSearch(
            FSSpec examples[],
            UInt32 numExamples,
            const FSSpec targetDirs[],
            UInt32 numTargets,
            UInt32 maxHits,
            UInt32 maxHitWords,
            Boolean allIndexes,
            Boolean includeRemote,
            FBCSearchSession* theSession);
```

`examples` is a pointer to an array of `FSSpec` records that refer to files. FBC will search for files that are similar to these files.

`numExamples` contains the number of `FSSpec` records in the array pointed to by examples.

`targetDirs` points to an array of `FSSpec` records referring to directories. If `targetDirs` is not `NULL` and `numTargets` is not zero, then only files residing in these directories will be included in the hit list returned by the search.

`targetDirs` points to an array of `FSSpec` records that refer to directories. If `numTargets` is zero, then this parameter can be set to `NULL`.

`numTargets` contains the number `FSSpec` records in the array pointed to by `targetDirs`.

`maxHits` the maximum number of hits that should be returned.

`maxHitWords` the maximum number of hit words that will be reported.

`includeRemote` is a `Boolean` value.

`theSession` points to a variable of type `FBCSearchSession` that will be created by this routine.

`FBCBlindExampleSearch` creates a new search session and conducts a similarity search using the files referred to in the array of `FSSpec` records provided in the examples parameter. If the `targetDirs` parameter is present (`numTargets` is not zero), then only files residing in the directories specified in `targetDirs` will be included in the hits found by the search. If `includeRemote` is true, then remote volumes will be included in the search session's list of target volumes.

If any of the example files are not indexed, then the search will proceed with the remainder of the files, and the error code `kFBCsomeFilesNotIndexed` will be returned. In this case, the search session will be created and a reference to it will be returned in `*theSession`.

Getting Information About Hits

Once a search is complete, a search session will contain a list of hits that were found during the search. The routines described in this section allow clients to access information about hits stored in a search session. Hit records are indexed 0 through count-1.

FBCGetHitCount

```
OSErr FBCGetHitCount(
            FBCSearchSession theSession,
            UInt32* count);
```

`theSession` is a pointer to a search session.

`count` is a pointer to a 32-bit integer.

`FBCGetHitCount` sets the variable pointed to by `count` to the number of hits in the search session. Hit records are indexed 0 through count-1.

FBCGetHitDocument

```
OSErr FBCGetHitDocument(
            FBCSearchSession theSession,
            UInt32 hitNumber,
            FSSpec* theDocument);
```

`theSession` is a pointer to a search session.

`hitNumber` is an index value referring to a hit record in the search session.

`theDocument` is a pointer to a `FSSpec` record.

`FBCGetHitDocument` returns the `FSSpec` record for the hit in the search session whose index is `hitNumber`.

FBCGetHitScore

```
OSErr FBCGetHitScore(
            FBCSearchSession theSession,
            UInt32 hitNumber,
            float* score);
```

`theSession` is a pointer to a search session.

`hitNumber` is an index value referring to a hit record in the search session.

`score` is a pointer to a variable of type `float`.

`FBCGetHitScore` returns relevance score assigned to the hit in the search session whose index is `hitNumber`. The score is a direct measure of the document's relevance to the search criteria in the context of this particular search. Scores are normalized to the range 0.0 - 1.0, and the most relevant hit from every search always has a score of 1.0.

FBCGetMatchedWords

```
OSErr FBCGetMatchedWords(
            FBCSearchSession theSession,
```

```
            UInt32 hitNumber,
            UInt32* wordCount,
            FBCWordList* list);
```

`theSession` is a pointer to a search session.

`hitNumber` is an index value referring to a hit record in the search session.

`wordCount` is a pointer to a 32-bit integer.

`list` is a pointer to a variable of type `FBCWordList`.

`FBCGetMatchedWords` returns a list of matched words for the hit in the search session whose index is `hitNumber`. This list of words illustrates why the hit was returned. On return, `*list` will contain a pointer to a word list structure and `*wordCount` will be set to the number of entries in that structure. Be sure to call `FBCDestroyWordList` to dispose of the structure when you are done with it.

The matched words for a hit are stored in the hit itself, so retrieving them is fast.

`FBCGetTopicWords`

```
OSErr FBCGetTopicWords(
            FBCSearchSession theSession,
            UInt32 hitNumber,
            UInt32* wordCount,
            FBCWordList* list);
```

`theSession` is a pointer to a search session.

`hitNumber` is an index value referring to a hit record in the search session.

`wordCount` is a pointer to a 32-bit integer.

`list` is a pointer to a variable of type `FBCWordList`.

`FBCGetTopicWords` returns a list of topical words for the hit in the search session whose index is `hitNumber`. This list of words provides a clue about "what the document is about." On return, `*list` will contain a pointer to a word list structure and `*wordCount` will be set to the number of entries in that structure. Be sure to call `FBCDestroyWordList` to dispose of the structure when you are done with it.

The list of topical words for a particular hit must be generated through the index file, so this call is significantly slower than `FBCGetMatchedWords`.

`FBCDestroyWordList`

```
OSErr FBCDestroyWordList(
            FBCWordList theList,
            UInt32 wordCount);
```

`theList` is a pointer to a word list.

`wordCount` is the number of words in the list.

`FBCDestroyWordList` disposes of a word list allocated by either `FBCGetMatchedWords` or

FBCGetTopicWords.

FBCReleaseSessionHits

```
OSErr FBCReleaseSessionHits(
            FBCSearchSession theSession);
```

theSession is a pointer to a search session. This session may contain hits generated by a search.

FBCReleaseSessionHits deallocates any information stored regarding hits from the last search from the search session. Volume configuration information is retained and once this call completes, the search session is ready to perform another search.

Summarizing Text

This call produces a summary containing the "most relevant" sentences found in the input text.

FBCSummarize

```
OSErr FBCSummarize(
            void* inBuf,
            UInt32 inLength,
            void* outBuf,
            UInt32* outLength,
            UInt32* numSentences);
```

inBuf points to the text to be summarized.

inLength is the length of the text pointed to by inBuf.

outBuf points to a buffer where the summary should be stored.

outLength is a pointer to a 32-bit integer. On input, this value is set to the size of the buffer pointed to by outBuf. On output, it is set to the actual length of the data stored in the buffer pointed to by outBuf.

numSentences is a pointer to a 32-bit integer. On input, this value is the maximum number of sentences desired in the summary. On output, it is set to the actual number of sentences generated. If numSentences is 0 on input, FBC takes the number of sentences in the input buffer and divides by 10. If the result is 0, then the value 1 is used as the maximum; otherwise, if the result is greater than 10, then the value 10 is used as the maximum.

Getting Information About Volumes

FBC provides the following utility routines for accessing information about volumes.

FBCVolumeIsIndexed

```
Boolean FBCVolumeIsIndexed (SInt16 theVRefNum);
```

`theVRefNum` is a volume reference number.

`FBCVolumeIsIndexed` returns `true` if the indicated volume has been indexed.

FBCVolumeIsRemote

```
Boolean FBCVolumeIsRemote(SInt16 theVRefNum);
```

`theVRefNum` is a volume reference number.

`FBCVolumeIsRemote` returns `true` if the indicated volume is located on a remote server. Clients may want to exclude networked volumes from searches to avoid network delays.

FBCVolumeIndexTimeStamp

```
OSErr FBCVolumeIndexTimeStamp(SInt16 theVRefNum,
            UInt32* timeStamp);
```

`theVRefNum` is a volume reference number .

`timeStamp` is a pointer to an unsigned 32 bit integer.

`FBCVolumeIndexTimeStamp` will return the time when the volume's index was last updated. The value returned in `timeStamp` is the same format as values returned by `GetDateTime`.

FBCVolumeIndexPhysicalSize

```
OSErr FBCVolumeIndexPhysicalSize(SInt16 theVRefNum,
            UInt32* size);
```

`theVRefNum` is a volume reference number .

`timeStamp` is a pointer to an unsigned 32 bit integer.

`FBCVolumeIndexPhysicalSize` returns the size of the volume's index file in bytes.

Reserving Heap Space

Clients of FBC can reserve space in their heap zone for their callback routine before conducting a search.

FBCSetHeapReservation

```
void FBCSetHeapReservation(UInt32 bytes);
```

`bytes` is an integer value containing the number of bytes that should be reserved.

`FBCSetHeapReservation` sets the number of bytes FBC should guarantee are available in the client application's heap whenever the client's call back routine is called during searches. If you do not explicitly reserve heap space by calling this routine, then 200K will be reserved for you.

Application-Defined Routine

Clients can provide a routine that will be called periodically during searches. This routine will provide clients with both information about the status of a search, and opportunity to cancel a search before it is complete.

Call back routines are defined as follows:

FBCCallbackProcPtr

```
typedef Boolean (*FBCCallbackProcPtr)(
            UInt16 phase,
            float percentDone,
            void *data);
```

`phase` is a 16-bit integer containing one of the following constants indicating the current status of the search:

```
enum
{
    kFBCphSearching             = 6,
    kFBCphMakingAccessAccessor  = 7,
    kFBCphAccessWaiting         = 8,
    kFBCphSummarizing           = 9,
    kFBCphIdle                  = 10,
    kFBCphCanceling             = 11
};
```

`percentDone` is a progress value in the range 0.0 - 1.0

`data` contains the same value provided to `FBCSetCallback` in the data parameter.

To avoid locking up the system while a search is in progress, the callback should either directly or indirectly call `WaitNextEvent`.

An ongoing search will be canceled if the call back function returns `true`.

FBCSetCallback

```
void FBCSetCallback(FBCCallbackProcPtr fn, void* data);
```

`fn` is a pointer to your call back function.

`data` is a value passed through to your call back function.

`FBCSetCallback` sets the call back function that will be called during searches. If a client does not define a call back function, then the default callback function is used. The default call back function calls `WaitNextEvent` and returns `false`.

Find By Content C Summary

Constants

```
enum
{
    gestaltFBCIndexingState      = 'fbci',
    gestaltFBCindexingSafe       = 0,
    gestaltFBCindexingCritical   = 1
};


enum
{
    gestaltFBCVersion            = 'fbcv',
    gestaltFBCCurrentVersion     = 0x0011
};

enum /* error codes */
{
    kFBCvTwinExceptionErr     = -30500,
                    /* miscellaneous error */
    kFBCnoIndexesFound        = -30501,
    kFBCallocFailed           = -30502,
                    /*probably low memory*/
    kFBCbadParam              = -30503,
    kFBCfileNotIndexed        = -30504,
    kFBCbadIndexFile          = -30505,
                    /*bad FSSpec, or bad data in file*/
    kFBCtokenizationFailed    = -30512,
                    /*couldn't read from document or query*/
    kFBCindexNotFound         = -30518,
    kFBCnoSearchSession       = -30519,
    kFBCaccessCanceled        = -30521,
    kFBCindexNotAvailable     = -30523,
    kFBCsearchFailed          = -30524,
    kFBCsomeFilesNotIndexed   = -30525,
    kFBCillegalSessionChange  = -30526,
                    /*tried to add/remove vols */
                    /*to a session  that has hits*/
    kFBCanalysisNotAvailable  = -30527,
    kFBCbadIndexFileVersion   = -30528,
    kFBCsummarizationCanceled = -30529,
    kFBCbadSearchSession      = -30531,
    kFBCnoSuchHit             = -30532
```

```
};

enum /* codes sent to the callback routine */
{
    kFBCphSearching             = 6,
    kFBCphMakingAccessAccessor  = 7,
    kFBCphAccessWaiting         = 8,
    kFBCphSummarizing           = 9,
    kFBCphIdle                  = 10,
    kFBCphCanceling             = 11
};
```

Data Types

```
    /* A collection of state information for searching*/
typedef struct OpaqueFBCSearchSession* FBCSearchSession;

    /* An ordinary C string (used for hit/doc terms)*/
typedef char* FBCWordItem;

    /* An array of WordItems*/
typedef FBCWordItem* FBCWordList;
```

Allocation and Initialization of Search Sessions

```
OSErr FBCCreateSearchSession(
            FBCSearchSession* searchSession);
OSErr FBCDestroySearchSession(
            FBCSearchSession theSession);
OSErr FBCCloneSearchSession(
            FBCSearchSession original,
            FBCSearchSession* clone);
```

Configuring Search Sessions

```
OSErr FBCAddAllVolumesToSession(
            FBCSearchSession theSession,
            Boolean includeRemote);
OSErr FBCSetSessionVolumes(
            FBCSearchSession theSession,
            const SInt16 vRefNums[],
            UInt16 numVolumes);
OSErr FBCAddVolumeToSession(
            FBCSearchSession theSession,
            SInt16 vRefNum);
OSErr FBCRemoveVolumeFromSession(
            FBCSearchSession theSession,
            SInt16 vRefNum);
OSErr FBCGetSessionVolumeCount(
            FBCSearchSession theSession,
            UInt16* count);
OSErr FBCGetSessionVolumes(
            FBCSearchSession theSession,
            SInt16 vRefNums[],
            UInt16* numVolumes);
```

Executing a Search

```
OSErr FBCDoQuerySearch(
            FBCSearchSession theSession,
            char* queryText,
```

```
                const FSSpec targetDirs[],
                UInt32 numTargets,
                UInt32 maxHits,
                UInt32 maxHitWords);
        OSErr FBCDoExampleSearch(
                FBCSearchSession theSession,
                const UInt32* exampleHitNums,
                UInt32 numExamples,
                const FSSpec targetDirs[],
                UInt32 numTargets,
                UInt32 maxHits,
                UInt32 maxHitWords);
        OSErr FBCBlindExampleSearch(
                FSSpec examples[],
                UInt32 numExamples,
                const FSSpec targetDirs[],
                UInt32 numTargets,
                UInt32 maxHits,
                UInt32 maxHitWords,
                Boolean allIndexes,
                Boolean includeRemote,
                FBCSearchSession* theSession);
```

Getting Information About Hits

```
        OSErr FBCGetHitCount(
                FBCSearchSession theSession,
                UInt32* count);
        OSErr FBCGetHitDocument(
                FBCSearchSession theSession,
                UInt32 hitNumber,
                FSSpec* theDocument);
        OSErr FBCGetHitScore(
                FBCSearchSession theSession,
                UInt32 hitNumber,
                float* score);
        OSErr FBCGetMatchedWords(
                FBCSearchSession theSession,
                UInt32 hitNumber,
                UInt32* wordCount,
                FBCWordList* list);
        OSErr FBCGetTopicWords(
                FBCSearchSession theSession,
                UInt32 hitNumber,
                UInt32* wordCount,
                FBCWordList* list);
        OSErr FBCDestroyWordList(
                FBCWordList theList,
                UInt32 wordCount);
        OSErr FBCReleaseSessionHits(
                FBCSearchSession theSession);
```

Summarizing Text

```
        OSErr FBCSummarize(
                void* inBuf,
                UInt32 inLength,
                void* outBuf,
                UInt32* outLength,
                UInt32* numSentences);
```

Getting Information About Volumes

```
Boolean FBCVolumeIsIndexed (SInt16 theVRefNum);
Boolean FBCVolumeIsRemote(SInt16 theVRefNum);
OSErr FBCVolumeIndexTimeStamp(SInt16 theVRefNum,
            UInt32* timeStamp);
OSErr FBCVolumeIndexPhysicalSize(SInt16 theVRefNum,
            UInt32* size);
```

Reserving Heap Space

```
void FBCSetHeapReservation(UInt32 bytes);
```

Application-Defined Routine

```
typedef Boolean (*FBCCallbackProcPtr)(
            UInt16 phase,
            float percentDone,
            void *data);
void FBCSetCallback(FBCCallbackProcPtr fn, void* data);
```

Downloadables

 Acrobat version of this Technote (129K).

Acknowledgments

Special thanks to David Casseres, Pete Gontier, Tim Holmes, Ingrid Kelly, Michael J. Kobb, Eric Koebler, Alice Li, and Wayne Loofbourrow.