Technote 1180

Sherlock's Find By Content Library

CONTENTS

Overview
Determining if Find By Content is Available
Working with Search Sessions
Setting up a Search Session
Performing Searches
Retrieving Information from a Search Session
Find By Content Reference
- Data Types
- Allocation and Initialization of Search Sessions
- Configuring Search Sessions
- Executing a Search
- Getting Information About Hits
- Summarizing Text
- Getting Information About Volumes
- Indexing Volumes, Folders, and Files
- Reserving Heap Space
- Application-Defined Routine

Find By Content C Summary

This Technote describes the Find by Content libraries used by Sherlock for searching the contents of files.

The Find by Content libraries export a full suite of routines and functions allowing applications to perform content based searches of files.

With MacOS 8.6, Text Extractor Plug-ins were introduced. These allow Find By Content to extract textual information from binary files for inclusion in index files. Text Extractor Plug-ins are documented in Technote TN1181, "Find by Content Text Extractor Plug-ins."

This Note is directed at application developers who wish to access the Find By Content library directly from their applications.

Overview

The Find By Content (FBC) facilities provided in Mac OS 8.6 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.

Back to top

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`.

Back to top

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.

Back to top

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.

```
/* 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;
}
```

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

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.

```
/* 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;
}
```

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

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.

Back to top

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.

```
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;
}
```

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

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.

```
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;
}
```

Listing 9. Searching a particular set of directories.

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`.

Back to top

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.

```
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;
}
```

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

Back to top

Back to top

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).

Back to top

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.

Back to top

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.

Back to top

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 pointer to 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 a pointer to 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.

Back to top

Executing a Search

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

FBCGetSessionVolumeCount

```
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`.

Back to top

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.

Back to top

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.

Back to top

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.

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

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

Back to top

Indexing Volumes, Folders, and Files

A new API has been added to Find By Content allowing for the immediate indexing of new or altered files. The new routine is declared as follows:

`FBCIndexItems`

```
OSErr FBCIndexItems(
     FSSpecArrayPtr theItems,
     UInt32 itemCount);
```

`theItems` is a pointer to an array of file specification records referring to the files to be indexed.

`itemCount` is the number of items in the array of file specification records.

`FBCIndexItems` indexes (or re-indexes) the files referred to in the array of file specification records passed as a pointer in the first parameter. If the volume containing a file already has an index, the document is added or re-indexed; and, if the volume does not contain an index, a new index is created.

Normally you will call `FBCIndexItems` after saving a file (or updating a file) on a volume containing an index. This will allow users to keep their indexes up to date without any additional effort. For more information about how to determine if a volume contains an index, refer to the Sherlock technote.

COMPATIBILITY NOTE
The symbol `FBCIndexItems` is not exported from the original version of the "Find By Content" shared library. Applications wishing to use this routine should weak link to this symbol and then test for it's presence before attempting to call it. Techniques for doing this are described in Technote TN1083, "Weak-Linking to a CFM-based Shared Library."

Back to top

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.

Back to top

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`.

Back to top

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);
```

Indexing files, folders, and volumes

```
OSErr FBCIndexItems(
            FSSpecArrayPtr theItems,
            UInt32 itemCount);
```

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);
```

Back to top

Further References

- Technote TN1141, "Extending and Controlling Sherlock"
- Technote TN1181, "Sherlock's Find by Content Text Extractor Plug-ins."

Back to top

Downloadables

 Acrobat version of this Note (K).

Back to top

To contact us, please use the Contact Us page.
Updated: 05-October-1999