This chapter describes how you can use the Dictionary Manager to create and work with dictionaries for input methods or other text services. The Dictionary Manager supplies a uniform and public dictionary format that lets you perform searching, insertion, and deletion.

Read this chapter if you are developing or enhancing an input method or other text service component that uses dictionaries.

To use this chapter, you should be familiar with the Macintosh script management system, the Text Services Manager, and parts of the File Manager. The script management system is described in the chapter "Introduction to Text on the Macintosh" in this book. The Text Services Manager is described in the chapter "Text Services Manager" in this book. The organization of the Dictionary Manager is based on B*-trees, used by the File Manager and the Finder. For more on the B*-trees, see the File Manager chapter of *Inside Macintosh: Files*.

This chapter presents a brief introduction to dictionaries and then discusses how you can make, access, locate records in, and modify them.

About Dictionaries for Input Methods

Input methods for 2-byte script systems use *dictionaries,* data files with information essential to the text conversions they perform. An input method uses its dictionary to convert the raw text entered by the user. For a discussion of raw text, conversion, and input methods, see the chapter "Text Services Manager" in this book.

Input methods commonly rely upon two or more dictionaries to perform conversion most efficiently. The **main dictionary** lists all standard conversion options for any valid syllabic or phonetic input. A main dictionary may have thousands to tens of thousands of entries, and is usually fixed in content. The **user dictionary,** also called an *editable dictionary,* is a complementary file in which users can add specialized or custom information that does not exist in the main dictionary. Because the main dictionaries of many input methods have only about 80 percent of the needed conversion options, a user dictionary is extremely valuable to users who customize the input process to improve its precision.

Users can also set dictionary learning. This allows the input method to incorporate frequency information as the user works, so that the frequency of combinations *in a particular grammatical context i*s taken into account in doing conversions. This makes a user dictionary even more valuable to the individual that has worked with it for a long time.

In principle, the dictionaries for different input methods of a given writing system should be very similar. For instance, most Japanese dictionaries contain information relevant to the conversion of Hiragana to Kanji. Korean dictionaries consist of data necessary for the conversion of Hangul to Hanja. Chinese dictionaries have entries relevant to the conversion of radical to Hanzi, Zhuyinfuhao to Hanzi, or Pinyin to Hanzi.

In practice, however, many currently available Chinese, Japanese, and Korean input methods use their own dictionary formats. Each input method has independently implemented operations to insert, delete, and search for the entries in its own dictionaries.

Input methods that use their own dictionary formats can understand only the dictionaries they create. This may be desirable for main dictionaries, because the features of a main dictionary can distinguish the quality of one input method from another; input method developers may be hesitant to share such dictionaries with other vendors. But for user dictionaries, incompatible formats create serious difficulties for users—particularly when a user dictionary contains many entries.

Consider the following situation. A user purchases an input method and uses it for perhaps a year, making numerous entries in the user dictionary. Then a new and better input method is introduced, but the new input method cannot understand the customized user dictionary. Because there is no general dictionary format, the user is forced to choose between two undesirable alternatives: creating an entirely new user dictionary by manually keying in thousands of previous entries, or continuing to use the old input method, forgoing the benefits of the new one.

This chapter describes a dictionary format that allows user dictionaries to be carried over from one input method to another, to avoid the difficulty just described. And although dictionaries are primarily of use to input methods, and are discussed in that context here, other text services such as thesauri or spelling checkers can also benefit from using dictionaries with this format.

About the Dictionary Manager

The Dictionary Manager supplies a uniform and public dictionary format and a set of operations that allows you to manipulate data in a dictionary file. This standard dictionary format helps to make the insertion, deletion, and searching operations in a dictionary available for all input methods.

This section describes the format and content of the data in a dictionary file, discusses the concept of garbage data and how the Dictonary Manager handles it, and presents some of the limitations you should be aware of before planning to use the Dictionary Manager.

Dictionary file types and Finder routing

Dictionaries belong in the Extensions folder within the user's System Folder. If your dictionary has a file type of `'dict'`, `'dic0'`, `'dic1'`, or `'dic2'`, the Finder automatically routes it to the Extensions folder if the user drops it on the System Folder. ◆

The Structure of a Dictionary

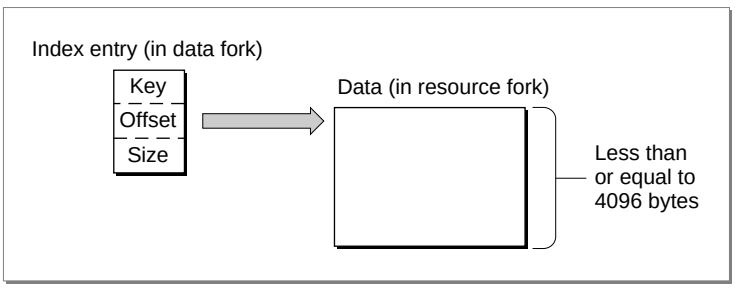
A **dictionary** is a collection of dictionary records. Each **dictionary record** consists of a key and some associated *data* referenced by that key. A **key** is a Pascal search string with a maximum length of 129 bytes (including the length byte). The data associated with a key has a maximum length of 4096 bytes.

The key for a dictionary record is stored separately from its data. The key, an offset to the data, and the length of the data make up the record's **index** entry. The index entry is stored as a **B*-tree** structure in the data fork of the dictionary file. The data is stored in the resource fork of the file; the Dictionary Manager accesses the data with Resource Manager partial resource routines. When a dictionary lookup is needed, the Dictionary Manager uses the key to find the location and size of the data in the resource fork. Then, it uses a partial resource reading to read the data into memory. (Routines for reading partial resources are described in the Resource Manager chapter of *Inside Macintosh: More Macintosh Toolbox*.) Figure 8-1 shows the general format of a dictionary record.

Note

Always use Dictionary Manager functions to gain access to records in the resource fork rather than examining them directly with Resource Manager routines. ◆

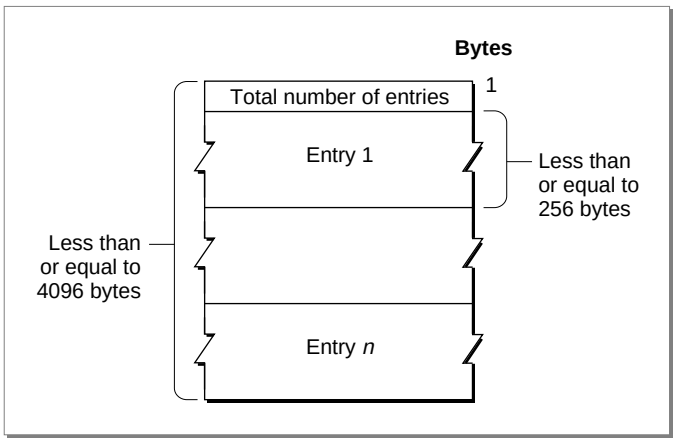
Figure 8-1 General format of a dictionary record



Each key-data pair is unique in a dictionary. No two keys in a single dictionary are identical.

Figure 8-2 shows the format of the data associated with a dictionary key. The first byte, which specifies the total number of entries in the data, is followed by a series of entries. Each entry has a maximum length of 256 bytes.

Figure 8-2 Format of data associated with a key

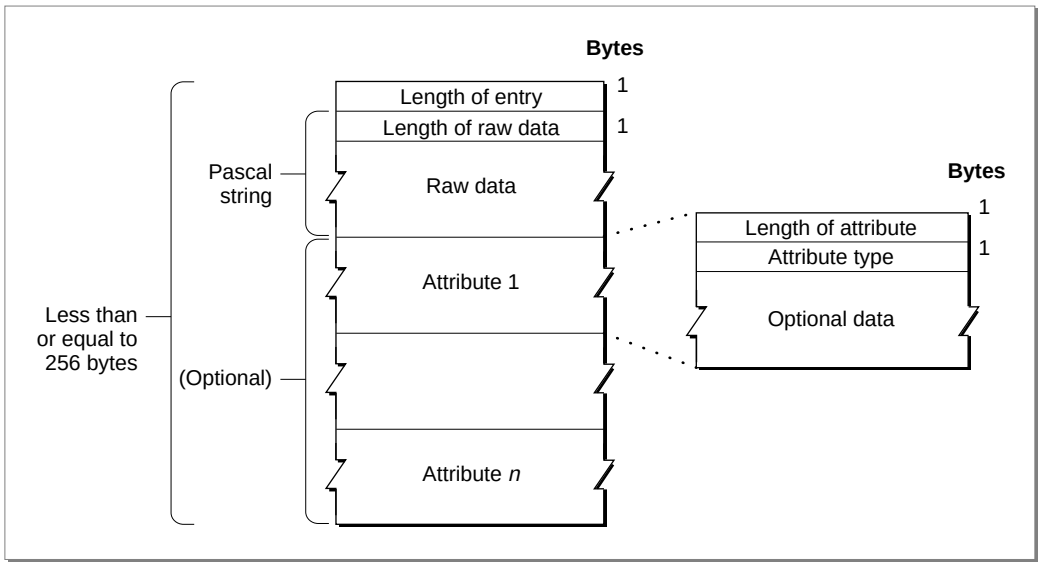


A **dictionary entry** in the data associated with a key contains raw data plus optional attributes**. Raw data** consists of any information related to the key entry. In a general dictionary it might be an explanation of the key; in a given East Asian dictionary it might be all the Chinese characters with the pronunciation of the key. A **data attribute** contains some information about the raw data—for example, grammatical or context-sensitive details, plus an attribute type. The **attribute type** is an integer constant in the range –128 to 127. The currently defined attribute types are listed on page 8-27.

Note
Apple reserves all negative attribute types. Positive attribute types are available for the use of developers of applications and text service components. ◆

Figure 8-3 shows the format of an entry. If data attributes are present, the first two bytes in each data attribute are the attribute size and the attribute type.

Figure 8-3 Format of an entry in the data associated with a key



Depending on the script system, the key, raw data, and attributes may differ.

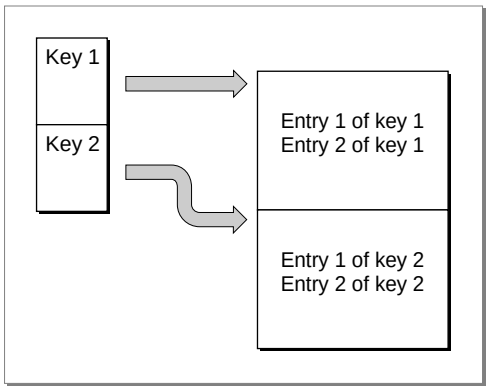
- In a record in an English dictionary, a key is any word; raw data is one or more definitions of the word; and a data attribute is the type of speech of the key—for example—verb, noun, adverb, adjective, or a combination of these.
- In a record in a Japanese dictionary, a key is a symbol of the phonetic subscript Hiragana; the associated raw data are the Kanji (ideographic characters); and the data attributes include the parts of speech or input method–specific attributes such as homonyms or groupings (clauses) of Kanji.
- In a record in a Chinese phonetic dictionary, a key is one of the phonetic symbols of Bopomofo; raw data is one or more Chinese words with the same pronunciation as the key; and there may be no associated data attributes.
- In a record in a Korean dictionary, a key may be a syllable or word in the Hangul subscript; associated raw data may be one or more Chinese words with the pronunciation of the key; and there may be no associated data attributes.

Garbage Data

In an editable dictionary, information is continually being added, deleted, or altered. Because it is too time-consuming to regenerate the entire dictionary each time a change is made, unused information called **garbage data** builds up over time. Garbage data is created whenever the size of the information associated with a key increases or decreases, or if the information is deleted. The data is no longer used by the dictionary.

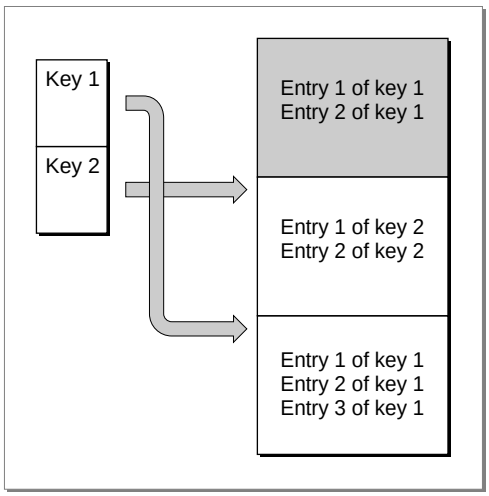
Consider the simple dictionary file in Figure 8-4. It has only two dictionary records; each record has two entries. There is no garbage data in either record.

Figure 8-4 A simple dictionary with no garbage data



With the addition of one entry to the first record, the Dictionary Manager allocates a new block at the end of the dictionary's resource fork to hold all the entries in the first record, and creates a new index entry that points to the new block. The data the old index entry points to is no longer accessible and becomes garbage data. See Figure 8-5.

Figure 8-5 Creating garbage data in a dictionary

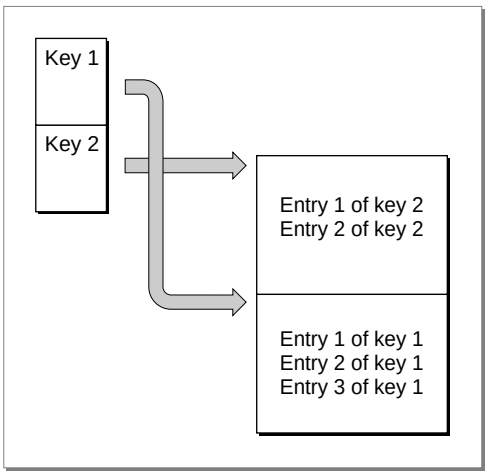


As the records in a dictionary file are modified, the size of the garbage data continues to increase. The Dictionary Manager keeps track of the amount of garbage data in a dictionary; to obtain the current size of garbage data in a dictionary, you can use the `GetDictionaryInformation` function (see page 8-24).

At some point, you may want to get rid of the garbage data permanently. The `CompactDictionary` function (see page 8-33) instructs the Dictionary Manager to create a new copy of the dictionary file, containing only valid entries. Once the new dictionary is constructed, the Dictionary Manager deletes the old one. (If the new dictionary fails to build properly, the original dictionary is preserved intact.) Note that `CompactDictionary` does not actually compress any data; it simply removes unusable information.

Figure 8-6 illustrates the structure of the simple dictionary built in Figure 8-4 and Figure 8-5 after the compaction process. Note that the order of the records in the resource fork may be different from what it was before compaction.

Figure 8-6 Deleting garbage data from a dictionary



Note
The Dictionary Manager creates new garbage data only if the size of the associated data is enlarged or reduced or if the associated data is deleted altogether. If you simply rearrange the order of the entries within a single record, without changing the size of the associated data, the Dictionary Manager does not create any garbage data. This feature is especially useful for input methods that support dictionary learning, in which the entries require constant rearrangement according to their frequency of use. ◆

Dictionary Manager Limitations

Consider these limitations before using the Dictionary Manager:

- The Dictionary Manager does not perform data compression. Your input method can compress part of the information before submitting it to the Dictionary Manager—but such compression would make your dictionary nontransferable and thus defeat a major purpose of using the Dictionary Manager.
- The Dictionary Manager utilizes partial resource reading and writing to manipulate the actual data in a dictionary. Hence, each dictionary may not exceed 16 MB, a Resource Manager limitation.

- The user cannot edit an active dictionary (one currently being used by an input method). This also is a Resource Manager limitation.
- If you are developing a sophisticated input method, you may decide not to convert your main dictionaries into the Dictionary Manager format, because you may not want to publicize the keys and associated data in your main dictionaries.
- It may not even be practical to convert your main dictionaries. For example, several input methods contain gigantic—and significantly compressed—main dictionaries. In such cases, the conversion and decompression of dictionaries into Dictionary Manager format might greatly increase the size of your dictionary.

In summary, the Dictionary Manager is best for constructing user dictionaries of moderate size. Nevertheless, it is possible and in some cases it may be practical to use the Dictionary Manager for a main dictionary also.

Using the Dictionary Manager

This section tells how to use the Dictionary Manager to create and manipulate dictionaries. Using Gestalt Manager, Dictionary Manager, and File Manager calls, you can

- determine whether the Dictionary Manager is present and what attributes it has
- make a new dictionary (create a file and initialize the dictionary)
- gain access to a dictionary (open it, close it , and get information about it)
- locate records in a dictionary (by key or by index)
- modify the contents of a dictionary (insert, replace, or delete information)
- compact a dictionary

Testing for the Presence of the Dictionary Manager

Use `Gestalt` with the `gestaltDictionaryMgrAttr` environment selector to obtain a result in the response parameter that identifies the attributes of the Dictionary Manager. A result of `gestaltDictionaryMgrPresent` (= 0) means that the Dictionary Manager is present.

For details on the `Gestalt` function, see the Gestalt Manager chapter in *Inside Macintosh: Operating System Utilities*.

Making a Dictionary

You make a new dictionary by first creating a file and then initializing it as a Dictionary Manager dictionary.

Creating the File

To create a dictionary file, you first use a File Manager function such as `FSpCreate` or `HCreate` to create a file. Listing 8-1 is a sample routine that creates a file for a user dictionary.

Listing 8-1 Creating a dictionary file

```
FUNCTION CreateUserDictionary (VAR dictionaryFSSpec: FSSpec;
                               creator, fileType: OSType;
                               script: ScriptCode): OSErr;
VAR
   err: OSErr;
   fileReply: StandardFileReply;
BEGIN
   err := noErr;

   {get dictionary name and filespec}
   StandardPutFile('Create empty dictionary as...',
                  'UserDictionary', fileReply);

   {delete existing dictionary if user OKs it}
   IF fileReply.sfGood THEN BEGIN
      dictionaryFSSpec := fileReply.sfFile;
      IF fileReply.sfReplacing THEN
         err := FSpDelete(dictionaryFSSpec);

      {create the empty dictionary file}
      IF err = noErr THEN BEGIN
         err := FSpCreate(dictionaryFSSpec, creator,
                          fileType, script);
         IF err <> noErr THEN
            DebugErrStr(err, 'FSpCreate');{handle error here}
      END
      ELSE
         DebugErrStr(err, 'FSpDelete');   {handle error here}
   END
   ELSE
      err := fnfErr;                       {assign error}
   CreateUserDictionary := err;
END;     {CreateUserDictionary}
```

Constructing the Dictionary

To make the internal structure of your newly created dictionary file, you use the `InitializeDictionary` function. You provide a file system specification pointer to the file you just created, you specify what maximum size the dictionary keys can have, and you can specify what search criteria—such as case-sensitivity—the dictionary will support.

The following code is a statement that initializes a dictionary file. It uses an application-defined constant (`kMaximumKeyLength`) to specify key length, an application-defined global (`gDictionaryScriptID`) to specify the script system for the dictionary, and the `kIsCaseSensitive` constant to specify that searches are to be case-sensitive.

```
err := InitializeDictionary(dictionaryFile, kMaximumKeyLength,
                            $1000 + kIsCaseSensitive,
                            gDictionaryScriptID);
```

Accessing a Dictionary

Before you can use a dictionary you must first open it. Once it is open, you can get information about it and you can use it. When you are finished with the dictionary, you must close it.

Opening and Closing the Dictionary

To open and use a dictionary, you must create an access path to the dictionary file using the `OpenDictionary` function. You provide a pointer to the file system specification record that defines the file, and you specify the read and write permission for the access path.

The `OpenDictionary` function returns a long integer, called a *dictionary reference number,* that specifies the open dictionary. You use that same dictionary reference number whenever you use the dictionary, and finally when you close the dictionary with the `CloseDictionary` function.

Listing 8-2 gives an example of how to create and close this access path. It consists of portions of the `CASE` statement from a sample application's menu-dispatching routine.

Listing 8-2 Opening and closing a dictionary file

```
{if user selects "Open dictionary" menu item:}
iOpenDictionary:
   IF gDictionaryReference = 0 THEN BEGIN
      {only open my own dictionary file types}
      fileTypes[0] := kMyDictionaryFileType;
```

```
            StandardGetFile(NIL, 1, fileTypes, fileReply);
            IF fileReply.sfGood THEN BEGIN
               gDictionaryFile := fileReply.sfFile;
               {open file with read-write permission}
               err := OpenDictionary(@gDictionaryFile,
                                     fsRdWrPerm,
                                     gDictionaryReference);
            END;
         END;

      {if user selects “Close dictionary” menu item:}
      iCloseDictionary:
         IF gDictionaryReference <> 0 THEN BEGIN
            err := CloseDictionary(gDictionaryReference);
            gDictionaryReference := 0;
         END;
```

Obtaining Information About the Dictionary

You can use the `GetDictionaryInformation` function to obtain the following information about a dictionary file:

- its file system specification record
- the number of records it contains
- the current size in bytes of its unused (garbage) data
- the script code of the script system it belongs to
- its maximum key length
- its search criteria

To identify the desired dictionary file, you use the dictionary reference number obtained when you open a dictionary file with the `OpenDictionary` function.

The `GetDictionaryInformation` function returns its information in a dictionary information record. The dictionary information record is defined by the `DictionaryInformation` data type as follows:

```
TYPE DictionaryInformation =
   RECORD
      dictionaryFSSpec:    FSSpec;        {file system spec}
      numberOfRecords:     LongInt;       {total no. of records}
      currentGarbageSize:  LongInt;       {size of unusable data}
      script:              ScriptCode;    {script system}
      maximumKeyLength:    Integer;       {maximum length of keys}
      keyAttributes:       UnsignedByte;  {key search criteria}
   END;
```

See page 8-25 for a complete description of these fields. Listing 8-3 shows a call to `GetDictionaryInformation` to obtain the number of records in a dictionary.

Listing 8-3 Obtaining information about a dictionary

```
FUNCTION GetNumberOfRecordsInDictionary(dictionaryReference:
                                          LONGINT): INTEGER;
VAR
   err:              OSErr;
   dictionaryInfo:   DictionaryInformation;
   numRecords:       Integer;
BEGIN
   numRecords := -1;          {return result in case of error}
   IF dictionaryReference <> 0 THEN BEGIN
      {get the dictionary information record}
      err := GetDictionaryInformation(gDictionaryReference,
                                      dictionaryInfo);
      IF err <> noErr THEN
         numRecords := dictionaryInfo.numberOfRecords
      ELSE
         DebugErrStr(err, 'GetDictionaryInformation');   {error}
   END;
   GetNumberOfRecordsInDictionary := numRecords;
END;  {GetNumberOfRecordsInDictionary}
```


Locating Records in a Dictionary

This section tells you how to use a dictionary—that is, how to extract records from it. You can obtain records from a dictionary in two general ways: by key (search string) and by index (position in the file).

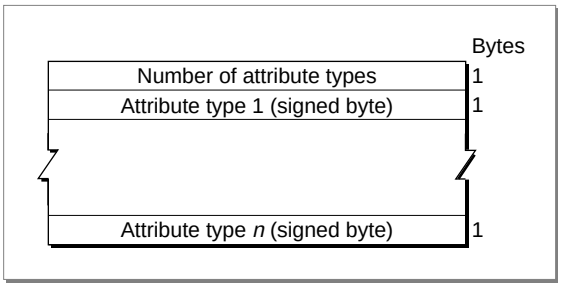
Locating Records by Key

You can use the `FindRecordInDictionary` function to search for dictionary records that match specified keys. Matching keys is perhaps the most standard dictionary search method: the user types in a key, and you search the dictionary for the data associated with that key.

You can provide a *requested attributes table* to narrow the search to only certain types of entries within the record that matches the search key. Figure 8-7 shows its format. You can request

- only the entries with the specified attributes
- the raw data of all the entries in the record without any attributes
- everything in the record

Figure 8-7 The requested attributes table



For example, you can use the requested attributes table to select only the verbs or nouns in the dictionary that match a key. The currently defined attribute types and their constants are listed in Table 8-2 on page 8-27.

Here is an example of how to use `FindRecordInDictionary`. Suppose the following entries are all the entries that match the key "hunch" in a dictionary record:

Raw data	Attribute type	Value	Optional attribute data
`'guess'`	`kNoun`	–1	
	`kVerb`	–2	
`'push'`	`kVerb`	–2	
	`kMyType1`	127	MyType1Data
`'bend'`	`kMyType2`	126	MyType2Data

Now suppose you call `FindRecordInDictionary` and pass a pointer to a requested attributes table that specifies two types: `kNoun` (–1) and `kMyType1` (127). `FindRecordInDictionary` returns the data shown in Table 8-1.

Table 8-1 Sample data returned by `FindRecordInDictionary`

Offset	Value	Explanation
00	2	Number of entries found
(first entry starts here)		
01	8	Length of entry
02	5	Length of raw data
03	`'guess'`	The raw data
08	1	Length of first attribute
09	–1	Attribute type (= `kNoun`)
(second entry starts here)		
10	18	Length of entry

Table 8-1 Sample data returned by `FindRecordInDictionary` (continued)

Offset	Value	Explanation
11	4	Length of raw data
12	`'push'`	The raw data
16	12	Length of first attribute
17	127	Attribute type (= `kMyType1`)
18	`'MyType1Data'`	Optional data for first attribute

Locating Records by Index

You can use the `FindRecordByIndexInDictionary` function to retrieve record data within a dictionary file by index rather than by matching key strings. In this way you can examine a specific record or sequence of records, to look for the information you need.

As with `FindRecordInDictionary`, you can provide a requested attributes table to narrow the search to certain types of entries. If you want to get all records with entries of a particular attribute type, you can call `FindRecordByIndexInDictionary` repeatedly. In Listing 8-4, the routine loops through the entire dictionary, displaying the key and the raw data of the first entry of each record in turn. (The application routine `GetIndexedDataStringFromRecord` converts the raw data from each record into a string for display.)

Listing 8-4 Displaying all records in a dictionary by index

```
PROCEDURE ShowAllEntries (dictionaryReference: LONGINT);
VAR
   err:                        OSErr;
   dictionaryInfo:             DictionaryInformation;
   index:                      Integer;
   keyString, descriptionStr: Str255;
   entriesHandle:              Handle;
   txtDialog:                  DialogPtr;
   finalTick:                  LongInt;
BEGIN
   IF dictionaryReference <> 0 THEN BEGIN
      {first find out how many records there are}
      err := GetDictionaryInformation(gDictionaryReference,
                                      dictionaryInfo);
      IF err = noErr THEN BEGIN
         entriesHandle := NewHandle(0);
         IF entriesHandle <> NIL THEN BEGIN
            descriptionStr := 'Displaying names in dictionary';
```

```
            txtDialog := ShowTextDialog(@descriptionStr[1],
                                        LENGTH(descriptionStr));
            Delay(60, finalTick);

            FOR index := 1 TO dictionaryInfo.numberOfRecords
            DO BEGIN
               {return raw data for all entries of each record}
               err := FindRecordByIndexInDictionary
                                        (dictionaryReference,
                                        index - 1, NIL,
                                        keyString, entriesHandle);

               {we only care about the first description string }
               GetIndexedDataStringFromRecord(entriesHandle, 1,
                                              descriptionStr);

               {format as "key: description"}
               keyString := CONCAT(keyString, ': ');
               keyString := CONCAT(keyString, descriptionStr);
               SetTextDialog(txtDialog, @keyString[1],
                             LENGTH(keyString));
               Delay(60, finalTick);
            END;
            CloseTextDialog(txtDialog);
            DisposeHandle(entriesHandle);
         END;
      END;
   END;
END;  {ShowAllEntries}
```

Modifying a Dictionary

This section tells you how to use the Dictionary Manager routines to add, replace, or delete dictionary records.

You can use the `InsertRecordToDictionary` function to add or replace a record in a specified dictionary. Because there cannot be two separate records with the same key value in a dictionary, adding a new record may nullify an existing one. To avoid such a problem, you can specify the **insertion mode,** which notifies the Dictionary Manager how you want the new record treated. The insertion mode determines whether to put the record into the dictionary

- only if it does *not* replace an existing record with the same key
- only if it *does* replace an existing record with the same key
- regardless of whether it adds a record or replaces another record

You can effectively insert and replace individual entries in records, in that you can obtain a record from a dictionary (with `FindRecordInDictionary` or `FindRecordByIndexInDictionary`), modify parts of the record, and then put the record back into the dictionary with `InsertRecordToDictionary`.

In Listing 8-5, the routine prompts the user for a key word and data for a new dictionary record. It then constructs the new record—in proper dictionary format—by calling the application routine `NewDictionaryEntry`. Finally, it calls `InsertRecordToDictionary`, providing a dictionary reference number to the desired dictionary file, a Pascal string representing the key, a handle to the new record, and a specification of how to insert the record into the dictionary.

Listing 8-5 Inserting a record into a dictionary

```
PROCEDURE AddNewRecord (dictionaryReference: LongInt);
VAR
   keyStr, descriptionStr: Str255;
   descriptionHandle:      Handle;
   err:                    OSErr;
BEGIN
   IF dictionaryReference <> 0 THEN BEGIN
      keyStr := Ask('Enter key:', '<key word>');
      IF keyStr <> '' THEN BEGIN
         descriptionStr := Ask(CONCAT(CONCAT
                              ('Enter description for "',
                              keyStr), '"'), '<record data>');

         IF descriptionStr <> '' THEN BEGIN
            descriptionHandle := NewDictionaryEntry
                                    (descriptionStr, 128, '');
            IF descriptionHandle <> NIL THEN BEGIN
               err := InsertRecordToDictionary
                                    (dictionaryReference,
                                    keyStr, descriptionHandle,
                                    kInsertOrReplace);

               IF err <> noErr THEN
                  DebugErrStr(err, 'InsertRecordToDictionary');
               DisposeHandle(descriptionHandle);
            END;
         END;
      END;
   END;
END;  {AddNewRecord}
```

To remove a record from a dictionary, call the `DeleteRecordFromDictionary` function. When you call `DeleteRecordFromDictionary` you specifiy the key of the record to be deleted and the dictionary reference number of the dictionary file that contains the record.

Remember that deleting records from a dictionary, or replacing them with shorter records, does not make the dictionary file any smaller. It simply creates garbage data.

Compacting a Dictionary

You can use the `CompactDictionary` function to reduce the size of the dictionary by removing garbage data from the dictionary file.

IMPORTANT
Before compacting a dictionary, be aware that the operation may require considerable time to complete. You should notify the user of this. Avoid the compaction operation unless it is absolutely necessary or is mandated by the user. ▲

Dictionary Manager Reference

This section describes the routines and related data structures and constants that are specific to the Dictionary Manager.

Data Structures

The `DictionaryInformation` data type, which defines the dictionary information record, is described with the `GetDictionaryInformation` function on page 8-24.

Routines

This section shows the functions for making, accessing, using, and modifying dictionaries.

Making a Dictionary

To make a dictionary file, first create a file with a File Manager function such as `FSpCreate` or `HCreate`, and then call the `InitializeDictionary` function described in this section.

InitializeDictionary

The `InitializeDictionary` function constructs, for the specified file, the internal B*-tree structure that makes it a dictionary file.

```
FUNCTION InitializeDictionary (theFSSpecPtr: FSSpecPtr;
                               maximumKeyLength: Integer;
                               keyAttributes: Byte;
                               script: ScriptCode): OSErr;
```

`theFSSpecPtr`
A pointer to a file system specification record. This record contains the filename, directory, and volume associated with this dictionary file.

`maximumKeyLength`
The maximum length of the keys in the dictionary, including the length byte. The length must be less than or equal to 129.

`keyAttributes`
The search criteria for the keys in the dictionary.

`script` The number that specifies the script system this dictionary supports.

DESCRIPTION

`InitializeDictionary` does not open the dictionary file after the Dictionary Manager initializes it. To open and use a dictionary file, use the `OpenDictionary` function (see page 8-22).

You can set the maximum key length of a dictionary only once; you cannot change it after the dictionary has been created. To maximize efficiency, keep the length to a minimum.

The `keyAttributes` parameter allows you to specify search criteria. For example, in one script system, it might be desirable to design the search to disregard case and be sensitive to diacritical marks, whereas in another script system these preferences might be reversed in keeping with the character encoding. Two predefined constants are available for the key attributes: the `kIsCaseSensitive` constant indicates that search procedures are to be case sensitive, and the `kIsNotDiacriticalSensitive` constant specifies that the search procedures are to ignore diacritical marks. To specify a combination of the different attributes, you add the constants together.

Constant	Value	Explanation
`kIsCaseSensitive`	16	Search is case-sensitive
`kIsNotDiacriticalSensitive`	32	Search is not diacritical-sensitive

SPECIAL CONSIDERATIONS

`InitializeDictionary` may move memory; your application should not call this function at interrupt time.

RESULT CODES

In addition to the standard File Manager, Memory Manager, and Resource Manager error codes, `InitializeDictionary` may return any of the following result codes.

`noErr`	0	No error
`btNoSpace`	–413	Insufficient disk space to store dictionary information
`keyLenErr`	–416	Maximum key length too great or equal to zero
`keyAttrErr`	–417	No such key attribute

SEE ALSO

Constants for all defined script codes are listed in the chapter "Script Manager" in this book.

File system specification records and File Manager error codes are described in *Inside Macintosh: Files*. Memory Manager error codes are described in *Inside Macintosh: Memory*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

Accessing a Dictionary

Once you have created and initialized a dictionary file, you can use the `OpenDictionary` and `CloseDictionary` functions to open and close the dictionary. Once the dictionary is open, you can get information about it with the `GetDictionaryInformation` function.

OpenDictionary

The `OpenDictionary` function creates an access path to the specified dictionary file.

```
FUNCTION OpenDictionary (theFSSpecPtr: FSSpecPtr;
                         accessPermission: SignedByte;
                         VAR dictionaryReference: LongInt)
                         : OSErr;
```

`theFSSpecPtr`
A pointer to the file system specification record for the file to open. This record contains the filename, directory, and volume associated with this dictionary file.

`accessPermission`
The read and write permission for the access path. This permission must follow the File Manager access permission conventions.

`dictionaryReference`
A number that specifies a particular open dictionary.

DESCRIPTION

The `OpenDictionary` function returns, in the `dictionaryReference` parameter, a dictionary reference number—an identifying value that you use to specify the dictionary in subsequent calls to the Dictionary Manager.

The data structures accessed through the `dictionaryReference` parameter are allocated in the current heap. If the same dictionary is to be shared across applications, make sure the current zone is the system zone, so the data structures will be allocated in the system heap.

The following constants define the allowed values for the `accessPermission` parameter:

Constant	Value	Explanation
`fsRdPerm`	1	Request read permission only
`fsWrPerm`	2	Request write permission only
`fsRdWrPerm`	3	Request exclusive read/write permission

If the requested permission is not granted, `OpenDictionary` returns a result code that specifies the type of error.

SPECIAL CONSIDERATIONS

`OpenDictionary` may move memory; your application should not call this function at interrupt time.

RESULT CODES

In addition to the standard File Manager, Memory Manager, and Resource Manager errors, `OpenDictionary` may return any of the following result codes.

`noErr`	0	No error
`notBTree`	–410	File is not a dictionary

SEE ALSO

File system specification records, File Manager access permissions, and File Manager error codes are described in *Inside Macintosh: Files*. Memory Manager error codes are described in *Inside Macintosh: Memory*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

For sample code that uses the `OpenDictionary` function, see Listing 8-2 on page 8-13.

CloseDictionary

The `CloseDictionary` function closes the specified open dictionary.

```
FUNCTION CloseDictionary (dictionaryReference: LongInt)
                         : OSErr;
```

`dictionaryReference`
A number that specifies a particular open dictionary.

RESULT CODES

In addition to the standard File Manager and Resource Manager errors, `CloseDictionary` may return any of the following result codes.

`noErr`	0	No error
`notBTree`	–410	File is not a dictionary

SEE ALSO

File Manager error codes are described in *Inside Macintosh: Files*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

For sample code that uses the `CloseDictionary` function, see Listing 8-2 on page 8-13.

GetDictionaryInformation

The `GetDictionaryInformation` function returns, in a dictionary information record, information about the specified dictionary.

```
FUNCTION GetDictionaryInformation
               (dictionaryReference: LongInt;
                VAR theDictionaryInformation:
                DictionaryInformation): OSErr;
```

`dictionaryReference`
A number that specifies a particular open dictionary.

`theDictionaryInformation`
Upon completion of the call, contains a filled-out dictionary information record that describes the dictionary.

DESCRIPTION

The `GetDictionaryInformation` function returns data about a dictionary in a dictionary information record in the `dictionaryInformation` parameter. The `DictionaryInformation` data type defines this record as follows:

```
TYPE DictionaryInformation =
   RECORD
      dictionaryFSSpec:    FSSpec;
      numberOfRecords:     LongInt;
      currentGarbageSize:  LongInt;
      script:              ScriptCode;
      maximumKeyLength:    Integer;
      keyAttributes:       UnsignedByte;
   END;
```

Field descriptions

`dictionaryFSSpec`
The file system specification record for this particular dictionary.

`numberOfRecords`
The number of records in the dictionary.

`currentGarbageSize`
The current size of unusable (garbage) information in the dictionary. For a discussion of garbage in a dictionary, see "Garbage Data" on page 8-8.

`script` The number that specifies the script system this dictionary supports.

`maximumKeyLength`
The maximum length of any key in the dictionary.

`keyAttributes` A value that specifies the criteria for key searching. For a description of the key attribute constants, see the description of the `InitializeDictionary` function on page 8-21.

SPECIAL CONSIDERATIONS

`GetDictionaryInformation` may move memory; your application should not call this function at interrupt time.

RESULT CODES

In addition to the standard File Manager and Resource Manager errors, `GetDictionaryInformation` may return any of the following result codes.

`noErr`	0	No error
`notBTree`	–410	File is not a dictionary

SEE ALSO

Constants for all defined script codes are listed in the chapter "Script Manager" in this book.

File Manager error codes are described in *Inside Macintosh: Files*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

For sample code that uses the `GetDictionaryInformation` function, see Listing 8-3 on page 8-15.

Locating Records in a Dictionary

The following section describes the Dictionary Manager functions that let you

- locate a record within a dictionary by its key
- locate a record within a dictionary by its index

You can constrain both key and index searches to include only entries with certain attributes.

FindRecordInDictionary

The `FindRecordInDictionary` function searches a dictionary for a record that matches the specified key, and returns entries with the specified attributes.

```
FUNCTION FindRecordInDictionary
                            (dictionaryReference: LongInt;
                             key: Str255;
                             requestedAttributeTablePointer: Ptr;
                             recordDataHandle: Handle): OSErr;
```

`dictionaryReference`
A number that specifies a particular open dictionary.

`key` A Pascal string that denotes the key to be matched.

`requestedAttributeTablePointer`
A pointer to a table with attributes that you can request. This parameter provides a way for you to narrow the search to specified types of entries in the record. For instance, you could use the requested attributes table to specify only the verbs in the record that matches the key.

`recordDataHandle`
On entry, any valid handle. Upon completion, a handle to the requested data.

DESCRIPTION

The `FindRecordInDictionary` function returns, in the `recordDataHandle` parameter, a handle to the record data: a collection of entries matching the key and the requested attributes. `FindRecordInDictionary` returns the data in standard Dictionary Manager data format—as shown in Figure 8-2 on page 8-6 and Figure 8-3 on page 8-7.

The Dictionary Manager uses the Memory Manager procedure `SetHandleSize` to set the size of the `recordDataHandle` parameter correctly. If the Dictionary Manager cannot change the size of the handle to accommodate the returned matched data, it returns a Memory Manager error.

To limit the search to specific types of attributes, you construct a requested attributes table and pass a pointer to that table to `FindRecordInDictionary`. The requested attributes table consists of a byte which specifies the number of attributes, followed by a list of attribute types, as shown in Figure 8-7 on page 8-16.

- If the `requestedAttributeTablePointer` parameter is `Ptr(-1)`, `FindRecordInDictionary` returns everything in the matching record (that is, both raw data and attributes for all entries in the record).
- If the `requestedAttributeTablePointer` parameter is `NIL`, `FindRecordInDictionary` returns the raw data of all the entries in the matching record, without any attached attributes.
- If the `requestedAttributeTablePointer` parameter is a valid pointer, `FindRecordInDictionary` returns only those entries in the matching record whose attributes match those in the requested attributes table. In this case, if a record in the dictionary has a key that matches the search key but no entries in the record possess the requested attributes, the returned `recordDataHandle` parameter references a data buffer one byte in length that contains a value of 0.

Table 8-2 lists constants for the currently defined attribute types.

Table 8-2 Defined attribute types for dictionary entries

Constant	Value	Explanation
`kNoun`	–1	Noun
`kVerb`	–2	Verb
`kAdjective`	–3	Adjective
`kAdverb`	–4	Adverb

SPECIAL CONSIDERATIONS

`FindRecordInDictionary` may move memory; your application should not call this function at interrupt time.

RESULT CODES

In addition to the standard File Manager, Memory Manager, and Resource Manager errors, `FindRecordInDictionary` can return any of the following result codes.

`noErr`	0	No error
`notBTree`	–410	File is not a dictionary
`btRecNotFnd`	–415	Record cannot be found
`btKeyLenErr`	–416	Key length too great or equal to zero

SEE ALSO

File Manager error codes are described in *Inside Macintosh: Files*. Memory Manager error codes are described in *Inside Macintosh: Memory*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

FindRecordByIndexInDictionary

The `FindRecordByIndexInDictionary` function locates a record in a dictionary by index, and returns entries with the specified attributes.

```
FUNCTION FindRecordByIndexInDictionary
                            (dictionaryReference: LongInt;
                             recordIndex: LongInt;
                             requestedAttributeTablePointer: Ptr;
                             VAR recordKey: Str255;
                             recordDataHandle: Handle): OSErr;
```

`dictionaryReference`
A number that specifies a particular open dictionary.

`recordIndex`
The index for the record to be searched; its position in the dictionary. The index range for `FindRecordByIndexInDictionary` is from 0 to one less than the maximum number of records in a dictionary. To obtain the maximum index range of a dictionary, you can use the `GetDictionaryInformation` function (see page 8-24).

`requestedAttributeTablePointer`
A pointer to a table with attributes that you can request. This parameter provides a way for you to narrow the search to specified types of entries in the record. For instance, you could use the requested attributes table to specify only the verbs in the record at that index.

`recordKey` Upon succcessful completion, contains the key of the indexed record.

`recordDataHandle`
A handle that contains a collection of entries in the indexed record that match the requested attributes.

DESCRIPTION

The `FindRecordByIndexInDictionary` function returns, in the `recordDataHandle` parameter, a handle to the record data: a collection of entries from the specified record matching the requested attributes.
`FindRecordByIndexInDictionary` returns the data in standard Dictionary Manager data format—as shown in Figure 8-2 on page 8-6 and Figure 8-3 on page 8-7.

The Dictionary Manager uses the Memory Manager procedure `SetHandleSize` to set the size of the `recordDataHandle` parameter correctly. If the Dictionary Manager cannot change the size of the handle to accommodate the returned matched data, it returns a Memory Manager error.

To limit the search to specific types of attributes, you construct a requested attributes table and pass a pointer to that table to `FindRecordByIndexInDictionary`. The requested attributes table and a list of defined attribute types are described with the `FindRecordInDictionary` function, on page 8-26.

- If the `requestedAttributeTablePointer` parameter is `Ptr(-1)`, `FindRecordByIndexInDictionary` returns everything in the matching record (that is, both raw data and attributes for all entries in the record).
- If the `requestedAttributeTablePointer` parameter is `NIL`, `FindRecordByIndexInDictionary` returns the raw data of all the entries in the matching record, without any attached attributes.
- If the `requestedAttributeTablePointer` parameter is a valid pointer, `FindRecordByIndexInDictionary` returns only those entries in the matching record whose attributes match those in the requested attributes table. In this case, if a record in the dictionary has a key that matches the search key but no entries in the record possess the requested attributes, the returned `recordDataHandle` parameter references a data buffer 1 byte in length that contains a value of 0.

SPECIAL CONSIDERATIONS

`FindRecordByIndexInDictionary` may move memory; your application should not call this function at interrupt time.

RESULT CODES

In addition to the standard File Manager, Memory Manager, and Resource Manager errors, `FindRecordByIndexInDictionary` may return any of the following result codes.

`noErr`	0	No error
`notBTree`	–410	File is not a dictionary
`btRecNotFnd`	–415	Record cannot be found

SEE ALSO

File Manager error codes are described in *Inside Macintosh: Files*. Memory Manager error codes are described in *Inside Macintosh: Memory*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

For sample code that uses the `FindRecordByIndexInDictionary` function, see Listing 8-4 on page 8-17.

Modifying a Dictionary

The routines described in this section allow you to modify the contents of a dictionary by adding, replacing, or deleting records.

InsertRecordToDictionary

The `InsertRecordToDictionary` function inserts a dictionary record into the specified dictionary file.

```
FUNCTION InsertRecordToDictionary
                            (dictionaryReference: LongInt;
                             key: Str255;
                             recordDataHandle: Handle;
                             whichMode: InsertMode): OSErr;
```

`dictionaryReference`
A number that specifies a particular open dictionary.

`key` A Pascal string that denotes the key of the record to be inserted.

`recordDataHandle`
A handle containing the data for the new record.

`whichMode` A value that determines whether the inserted record is to replace a record in the dictionary whose key matches the `key` parameter.

DESCRIPTION

The `InsertRecordToDictionary` function places the specified record into the specified dictionary. The `recordDataHandle` parameter must be a handle to data formatted like the data of a dictionary record, as shown in Figure 8-2 on page 8-6. Each entry in the data must be formatted as shown in Figure 8-3 on page 8-7. If the data size referenced by the `recordDataHandle` parameter exceeds the maximum of 4096 bytes, `InsertRecordToDictionary` returns a `recordDataTooBigErr` result code.

The `whichMode` parameter controls the insertion mode, the manner in which the insertion can take place. There are three possibilities, for which the Dictionary Manager defines three constants:

Constant	Value	Description
`kInsert`	0	Insert the record only if no existing record has a matching key. If a record with a matching key already exists in the dictionary, this function returns the result code `btDupRecErr`.
`kReplace`	1	Insert the record only if it replaces an existing record with a matching key. If no existing record in the dictionary has a matching key, this function returns the result code `btRecNotFnd`.
`kInsertOrReplace`	2	Insert the new record either way. Add it if no existing record in the dictionary has a matching key; replace the existing record if there is a match.

If `InsertRecordToDictionary` returns one of the errors listed in "Result Codes," the specified record was not inserted or replaced.

SPECIAL CONSIDERATIONS

`InsertRecordToDictionary` may move memory; your application should not call this function at interrupt time.

RESULT CODES

In addition to the standard File Manager, Memory Manager, and Resource Manager errors, `InsertRecordToDictionary` can return one of the following result codes.

`noErr`	0	No error
`notBTree`	–410	File is not a dictionary
`btNoSpace`	–413	Insufficient disk space to store dictionary
`btDupRecErr`	–414	Record already exists
`btRecNotFnd`	–415	Record cannot be found
`btKeyLenErr`	–416	Key length too great or equal to zero
`unknownInsertModeErr`	–20000	No such insertion mode
`recordDataTooBigErr`	–20001	Entry data bigger than buffer size

SEE ALSO

File Manager error codes are described in *Inside Macintosh: Files*. Memory Manager error codes are described in *Inside Macintosh: Memory*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

For sample code that uses the `InsertRecordToDictionary` function, see Listing 8-5 on page 8-19.

DeleteRecordFromDictionary

The `DeleteRecordFromDictionary` function removes a record from the specified dictionary file.

```
FUNCTION DeleteRecordFromDictionary
                    (dictionaryReference: LongInt;
                       key: Str255): OSErr;
```

`dictionaryReference`
A number that specifies a particular open dictionary.

`key` A Pascal string that denotes the key of the record to be deleted.

DESCRIPTION

If `DeleteRecordFromDictionary` returns any of the errors listed in "Result Codes," it did not remove any records from the specified dictionary.

SPECIAL CONSIDERATIONS

`DeleteRecordFromDictionary` may move memory; your application should not call this function at interrupt time.

RESULT CODES

In addition to the standard File Manager, Memory Manager, and Resource Manager errors, `DeleteRecordFromDictionary` may return any of the following result codes.

`noErr`	0	No error
`notBTree`	–410	File is not a dictionary
`btRecNotFnd`	–415	Record cannot be found
`btKeyLenErr`	–416	Key length too great or equal to zero

SEE ALSO

File Manager error codes are described in *Inside Macintosh: Files*. Memory Manager error codes are described in *Inside Macintosh: Memory*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

Compacting a Dictionary

The routine described in this section allows you to compact a dictionary file.

CompactDictionary

The `CompactDictionary` function compacts the specified dictionary file by removing all garbage data from it.

```
FUNCTION CompactDictionary (dictionaryReference:LongInt)
                            : OSErr;
```

`dictionaryReference`
A number that specifies a particular open dictionary.

DESCRIPTION

The `CompactDictionary` function removes garbage data by creating a new copy of the dictionary file that contains only valid entries. Once the new dictionary is constructed, the Dictionary Manager deletes the old one.

If there is insufficient disk space to build the new dictonary, `CompactDictionary` returns the `btNoSpace` error message, and the original dictionary is preserved intact.

Note that `CompactDictionary` makes a dictionary file smaller by removing unusable information. It does not actually compress any data.

SPECIAL CONSIDERATIONS

`CompactDictionary` may move memory; your application should not call this function at interrupt time.

RESULT CODES

In addition to the standard File Manager, Memory Manager, and Resource Manager errors, `CompactDictionary` may return any of the following result codes.

`noErr`	0	No error
`notBTree`	–410	File not a dictionary
`btNoSpace`	–413	Insufficient disk space to store dictionary information

SEE ALSO

File Manager error codes are described in *Inside Macintosh: Files*. Memory Manager error codes are described in *Inside Macintosh: Memory*. Resource Manager error codes are described in *Inside Macintosh: More Macintosh Toolbox*.

Summary of the Dictionary Manager

Pascal Summary

Constants

```
CONST    {Data Insertion Modes}
   kInsert              = 0;  {only insert input entry if nothing in }
                              { dictionary matches key}
   kReplace             = 1;  {only replace entries that match key with }
                              { input entry}
   kInsertOrReplace     = 2;  {insert entry if nothing in the dictionary }
                              { matches key; if already matched }
                              { entries exist, replace them with the }
                              { input entry}

CONST {Key Attribute Constants}
   kIsCaseSensitive           = 16;    {diacritical mark is case sensitive}
   kIsNotDiacriticalSensitive = 32;    {diacritical mark not case sensitive}

CONST {Registered Attribute Types}
   kNoun       =  -1;   {noun}
   kVerb       =  -2;   {verb}
   kAdjective  =  -3;   {adjective}
   kAdverb     =  -4;   {adverb}
```

Data Types

```
TYPE
   InsertMode = Integer;

   AttributeType = Integer:
```

```
DictionaryInformation =
   RECORD
      dictionaryFSSpec:    FSSpec;        {file system specification }
                                          { record for this dictionary}
      numberOfRecords:     LongInt;       {number of records in }
                                          { this dictionary}
      currentGarbageSize:  LongInt;       {current size of garbage }
                                          { (unusable) data in dictionary}
      script:              ScriptCode;    {script system supported by }
                                          { this dictionary}
      maximumKeyLength:    Integer;       {maximum length of any key }
                                          { in this dictionary}
      keyAttributes:       UnsignedByte;  { key search criteria}
   END;
```


Routines

Making a Dictionary

```
FUNCTION InitializeDictionary
                           (theFSSpecPtr: FSSpecPtr;
                            maximumKeyLength: Integer;
                            keyAttributes: Byte;
                            script: ScriptCode): OSErr;
```

Accessing a Dictionary

```
FUNCTION OpenDictionary    (theFSSpecPtr: FSSpecPtr;
                            accessPermission: SignedByte;
                            VAR dictionaryReference: LongInt):
                            OSErr;
FUNCTION CloseDictionary   (dictionaryReference: LongInt):
                            OSErr;
FUNCTION GetDictionaryInformation
                           (dictionaryReference: LongInt;
                            VAR theDictionaryInformation:
                            DictionaryInformation): OSErr;
```

Locating Records in a Dictionary

```
FUNCTION FindRecordInDictionary
                           (dictionaryReference: LongInt;
                            key: Str255;
                            requestedAttributeTablePointer: Ptr;
                            recordDataHandle: Handle): OSErr;
```

```
FUNCTION FindRecordByIndexInDictionary
                            (dictionaryReference: LongInt;
                             recordIndex: LongInt;
                             requestedAttributeTablePointer: Ptr;
                             VAR recordKey: Str255;
                             recordDataHandle: Handle): OSErr;
```

Modifying a Dictionary

```
FUNCTION InsertRecordToDictionary
                            (dictionaryReference:LongInt;
                             key: Str255;
                             recordDataHandle: Handle;
                             whichMode: InsertMode): OSErr;
FUNCTION DeleteRecordFromDictionary
                            (dictionaryReference: LongInt;
                             key: Str255): OSErr;
```

Compacting a Dictionary

```
FUNCTION CompactDictionary  (dictionaryReference:LongInt)
                             OSErr;
```

C Summary

Constants

```
/* Dictionary data insertion modes. */
enum {
 kInsert = 0,              /* Only insert the input entry if there is nothing
                              in the dictionary that matches the key. */
 kReplace = 1,             /* Only replace the entries which match the key
                                 with the input entry. */

 kInsertOrReplace = 2      /* Insert the entry if there is nothing in the
                              dictionary which matches the key. If there are
                              already matched entries, replace the existing
                              matched entries with the input entry. */
};

/* Key attribute constants. */
#define kIsCaseSensitive 0x10               /* case-sensitive = 16 */
#define kIsNotDiacriticalSensitive 0x20     /* non-diac-sensitive = 32 */
```

```
/* Registered attribute type constants.*/
enum {
 kNoun = -1,
 kVerb = -2,
 kAdjective = -3,
 kAdverb = -4
};
```


Data Types

```
typedef short InsertMode;

typedef short AttributeType;

/* Dictionary information record.*/
struct DictionaryInformation{
   FSSpec         dictionaryFSSpec;
   long           numberOfRecords;
   long           currentGarbageSize;
   ScriptCode     script;
   short          maximumKeyLength;
   unsigned char  keyAttributes;
};
typedef struct DictionaryInformation DictionaryInformation;
```

Routines

Making a Dictionary

```
pascal OSErr InitializeDictionary
                            (FSSpecPtr theFsspecPtr,
                             short maximumKeyLength,
                             unsigned char keyAttributes,
                             ScriptCode script)
```

Accessing a Dictionary

```
pascal OSErr OpenDictionary (FSSpecPtr theFsspecPtr,
                             char accessPermission,
                             long *dictionaryReference)
pascal OSErr CloseDictionary
                            (long dictionaryReference)
```

```
pascal OSErr GetDictionaryInformation
                            (long dictionaryReference,
                             DictionaryInformation
                             *theDictionaryInformation)
```

Locating Records in a Dictionary

```
pascal OSErr FindRecordInDictionary
                            (long dictionaryReference, ConstStr255Param key,
                             Ptr requestedAttributeTablePointer,
                             Handle recordDataHandle)
pascal OSErr FindRecordByIndexInDictionary
                            (long dictionaryReference,
                             long recordIndex,
                             Ptr requestedAttributeTablePointer,
                             Str255 recordKey, Handle recordDataHandle)
```

Modifying a Dictionary

```
pascal OSErr InsertRecordToDictionary
                            (long dictionaryReference,
                             ConstStr255Param key, Handle recordDataHandle,
                             InsertMode whichMode)
pascal OSErr DeleteRecordFromDictionary
                            (long dictionaryReference, ConstStr255Param key)
```

Compacting a Dictionary

```
pascal OSErr CompactDictionary
                            (long dictionaryReference)
```

Assembly-Language Summary

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
`InitializeDictionary`	`_InitializeDictionary`
`OpenDictionary`	`_OpenDictionary`
`CloseDictionary`	`_CloseDictionary`
`InsertRecordToDictionary`	`_InsertRecordToDictionary`
`DeleteRecordFromDictionary`	`_DeleteRecordFromDictionary`
`FindRecordInDictionary`	`_FindRecordInDictionary`
`FindRecordByIndexInDictionary`	`_FindRecordByIndexInDictionary`
`GetDictionaryInformation`	`_GetDictionaryInformation`
`CompactDictionary`	`_CompactDictionary`

Result Codes

`notBTree`	–410	File not a dictionary
`btNoSpace`	–413	Insufficient disk space to store dictionary information
`btDupRecErr`	–414	Record already exists
`btRecNotFnd`	–415	Record cannot be found
`btKeyLenErr`	–416	Key length too great or equal to zero
`btKeyAttrErr`	–417	Dictionary Manager doesn't understand an attribute
`unknownInsertModeErr`	–20000	No such insertion mode
`recordDataTooBigErr`	–20001	Entry data bigger than buffer size
`invalidIndexErr`	–20002	Invalid index