This chapter describes the Collection Manager, which provides an abstract data type you can use to store collections of information. Read this chapter if you need to work with some advanced features of QuickDraw GX printing, including print dialog boxes, or if you want to create collections for purposes specific to your application.

Before reading this chapter, you might want to familiarize yourself with the information in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. For some examples of how the Collection Manager is used by other parts of QuickDraw GX, you should read the chapter "Page Formatting and Dialog Box Customization" in *Inside Macintosh: QuickDraw GX Printing*.

This chapter introduces the collection object as an abstract data type and describes the properties of this object. It then shows how to use the functions provided by the Collection Manager to

- create and manipulate collection objects
- add information to a collection object
- retrieve information from a collection object
- store a collection object to disk and retrieve a collection object from disk

This chapter also contains reference information for all data types, functions, and resources associated with the Collection Manager.

About the Collection Manager

The Collection Manager implements an abstract data type that allows you to store multiple pieces of related information. This abstract data type is called a collection object.

Collection Objects

A **collection object,** or simply a **collection,** is an abstract data type that allows you to store information. A collection is like an array in that it contains a number of individually accessible items. However, a collection offers some advantanges over an array:

- A collection allows for a variable number of data items. You can add items to a collection or remove items from a collection during run time, and the Collection Manager automatically resizes the collection.
- A collection allows for variable-size items. Each item in a collection can contain data of any size.

There are some corresponding disadvantages to using a collection versus using an array:

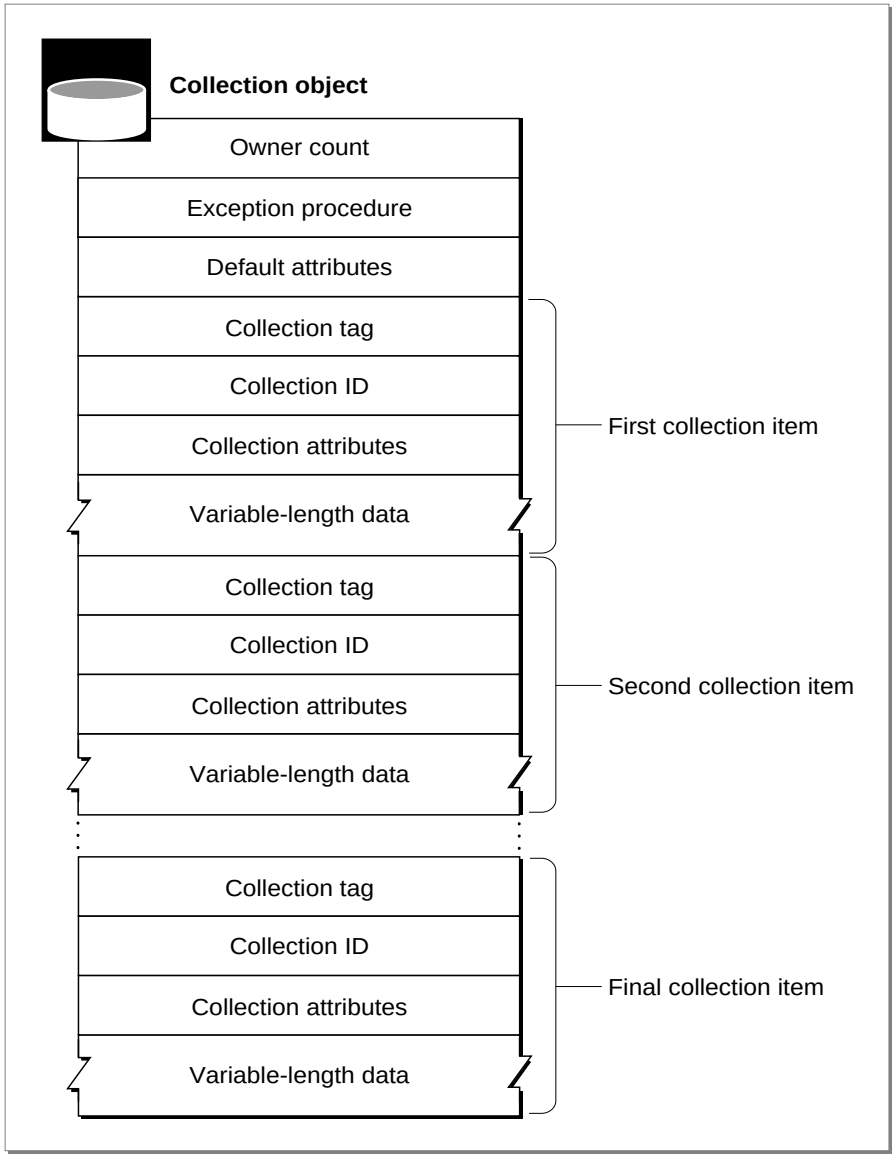
- You must store and retrieve information in a collection using Collection Manager functions, which is not as efficient as accessing an item of an array.
- The Collection Manager stores extra information about the collection and about each item in the collection, so a collection requires more memory than a comparable array.

A collection is also similar to a database, in that you can store information and retrieve it using a variety of search mechanisms. However, a collection has many more limitations than a real database. For example, the Collection Manager provides only a few mechanisms for searching a collection. Also, a collection is entirely memory-based. You can use a collection only when the entire contents of the collection are in memory, which makes a collection more like a powerful array than a database.

The internal structure of a collection object is private—you must store information in a collection and retrieve information from it by providing a Collection Manager function with a reference to the collection.

Figure 5-1 depicts the accessible properties of a collection object. Note that, because a collection is an object and not a public data structure, the order of the properties as shown is completely arbitrary.

Figure 5-1 The collection object



As Figure 5-1 shows, a collection object contains

- an **owner count,** which reflects the current number of references to the collection
- an exception procedure, which you can use to handle errors that occur while operating on the collection
- default attributes, which are described in "Collection Attributes" beginning on page 5-9.
- a number of collection items, which are described in the next section

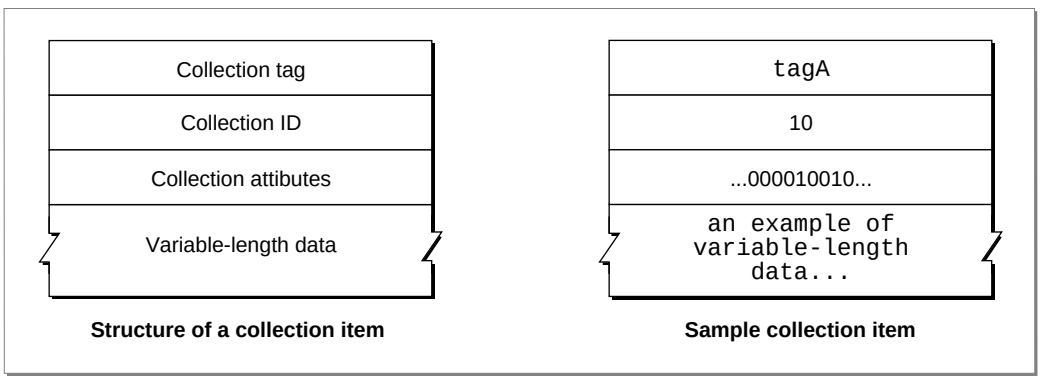
The Collection Manager maintains the owner count for you, although you can increment or decrement it by cloning or disposing of the collection, as described in “Creating or Disposing of a Collection” beginning on page 5-14 and “Cloning or Copying a Collection” beginning on page 5-14.

The Collection Manager allows you to install an **exception procedure** for each collection object. When the Collection Manager is operating on a collection and an error occurs, the Collection Manager calls the collection’s exception procedure (if you installed one) and passes to it the result code associated with the error that occurred. Your exception procedure can then respond to the error. For more information about exception procedures, see “Getting and Setting the Exception Procedure for a Collection” beginning on page 5-58 and the description of an application-defined exception procedure on page 5-101.

Collection Items

A collection is composed of **collection items.** Figure 5-2 shows the general structure of a collection item and also shows a sample collection item. Note that, because a collection item is always part of a collection object, you always access the information in a collection item using Collection Manager functions. Therefore, the order of the properties shown in Figure 5-2 is completely arbitrary.

Figure 5-2 The collection item



As Figure 5-2 shows, each collection item contains these properties:

- **Collection tag.** A collection tag is a four-character identifier that, in conjunction with the collection ID, uniquely identifies the collection item.
- **Collection ID.** A collection ID is a `long` value that, in conjunction with the collection tag, uniquely identifies the collection item.

- **Collection attributes.** The collection attributes are a set of 32 bit flags, each of which represents an attribute of the collection item. The Collection Manager defines the meaning of some of these attributes and leaves some of them for you to define. See the next section for more information about collection attributes.
- **Variable-length data.** The variable-length data contains the actual data of the item. This data corresponds to the contents of an item in an array, except items in the same collection can store data of different sizes, whereas items in a single array must all store data of the same size.

The Collection Manager uses a collection item's collection tag and collection ID to uniquely identify the item. As an example, in any collection there can be exactly one item with a collection tag of `'tagA'` and a collection ID of 10.

When you want to retrieve the data stored in an item, you can specify the item by providing a Collection Manager function with the item's collection tag and collection ID. You can also specify a collection item using one of the other methods provided by the Collection Manager. See "Methods of Identifying Collection Items" beginning on page 5-11 for more information.

Collection Attributes

Each collection item has 32 **attributes.** Each attribute is represented by one bit flag in the item's attributes property. Therefore each attribute is either set or clear. An item's attributes are stored in a 32-bit long word. The bits are numbered 0 through 31. Bit 31 is the high bit.

The upper 16 bits of an item's attributes property represent attributes that are reserved for use by Apple Computer, Inc. Currently, two of these attributes are defined:

- Bit 31 represents the **lock attribute.** When an item has this attribute set, attempts to replace the item result in an error. When this attribute is clear, you can replace the item.
- Bit 30 represents the **persistence attribute.** When an item has this attribute set, the Collection Manager includes this item when flattening the collection. When this attribute is clear, the Collection Manager ignores the item when flattening the collection. See "Flattening and Unflattening a Collection" beginning on page 5-37 for more information about flattening collections.

The other 14 reserved attribute bits are called the **reserved attributes.**

The lower 16 bits of an item's attributes property represent attributes that you can define for purposes suitable to your application. For example, you could use one of these attributes to mark all of the items that you wanted to write to disk and remove from the collection should you need more memory. These 16 attributes are called the **user attributes.**

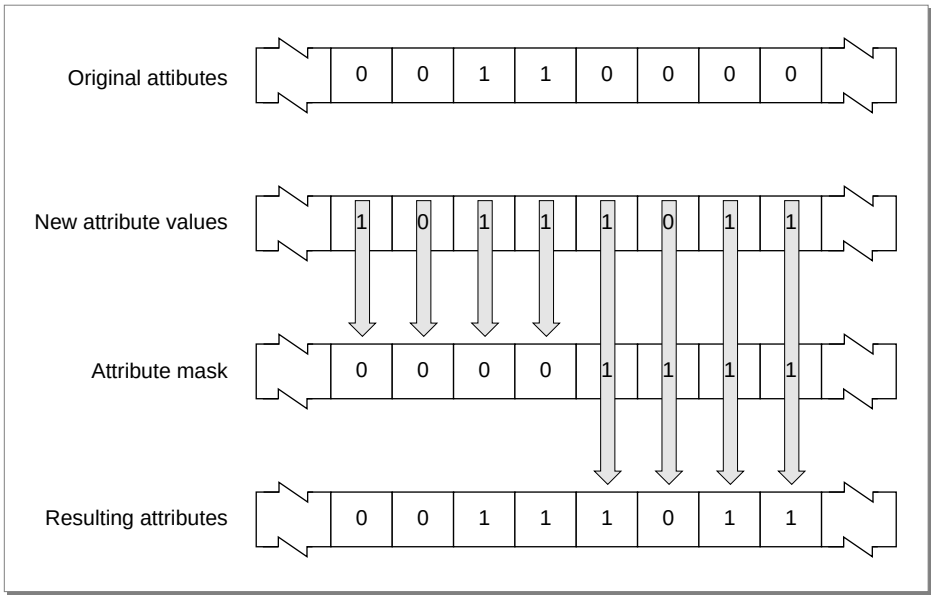
Depending on your application, you can set and examine the user attributes individually, or you can set and examine combinations of them. As an example, if your application uses collections that contain four distinct types of items, you could combine two user attributes to provide the four values (0–3) necessary to identify the four types of items.

Every collection object contains **default attributes.** A collection's default attributes determine the initial attribute values assigned to items added to that collection. For example, you could set the lock and persistence default attributes for a collection. From then on, when you added an item to the collection, the new item would have its lock and persistence attributes set. Of course, you would still be free to edit the attributes for the new item after adding it to the collection.

The Collection Manager provides a mechanism for editing attributes that allows you to set (or clear) the values of some attributes while leaving the values of other attributes alone. To edit attributes, you provide an **attribute mask,** in which you specify the attributes you want to edit, and new attribute values, in which you specify the new values for the attributes.

Figure 5-3 depicts this editing mechanism.

Figure 5-3 Editing attributes in a collection item



When editing an item's attributes, you provide an attribute mask and new attribute values. For every attribute:

- If you set the corresponding bit of the attribute mask to 0, the Collection Manager leaves the attribute unchanged from the original. The new value for the attribute (which you provide in the new attribute values) is ignored.
- If you set the corresponding bit in the attribute mask to 1, the Collection Manager copies the new attribute value you provide for this attribute. The original value of this attribute is overwritten.

You use this mechanism when editing an item's attributes, when editing a collection's default attributes, when searching for items whose attributes match a certain pattern, when flattening parts of a collection, and when purging items from a collection. For an example, see "Changing the Default Attributes of a Collection" beginning on page 5-15.

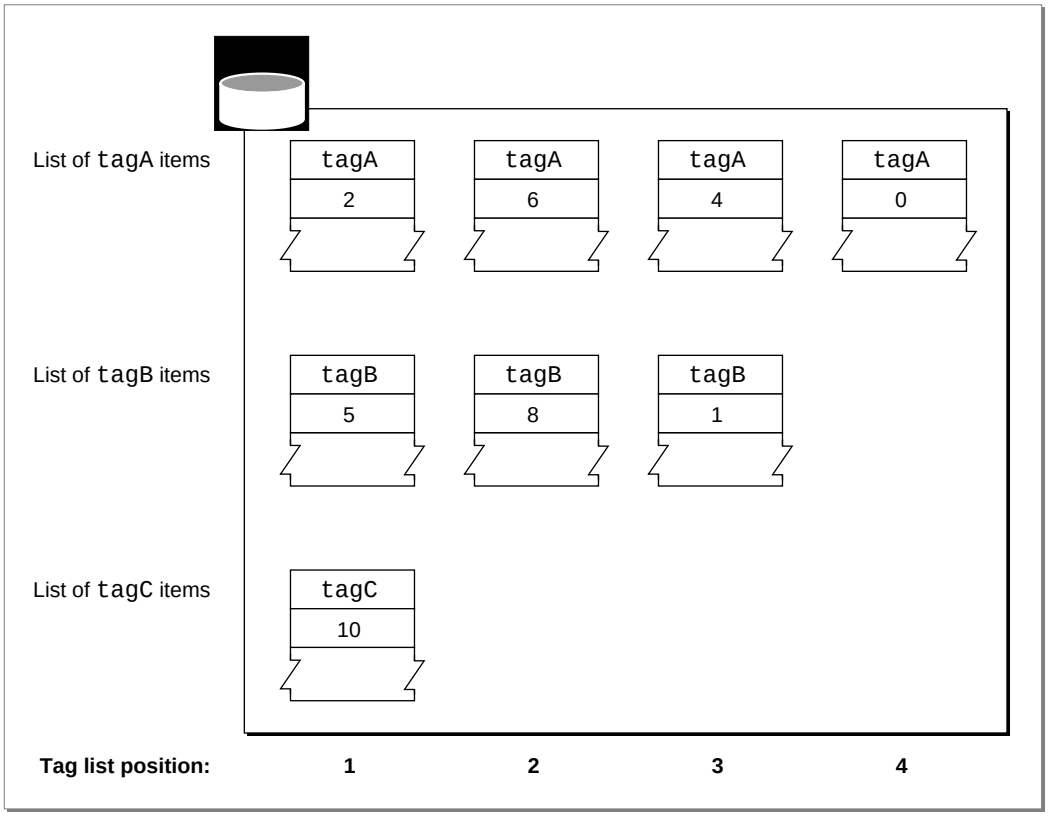
Methods of Identifying Collection Items

Many Collection Manager functions operate on an individual item within a collection. For example, the Collection Manager provides functions that allow you to replace the variable-length data for a particular item as well as functions that allow you to retrieve an item's variable-length data.

When calling these Collection Manager functions, you need to specify which collection item you want to examine or edit. The Collection Manager provides three methods of specifying a particular item in a collection:

- The collection tag and the collection ID. Together, these two properties uniquely identify an item in a collection. The collection IDs of collection items with the same collection tag do not have to be sequential. For example, the collection shown in Figure 5-4 has four items with the `'tagA'` collection tag. These four items have collection IDs 2, 6, 4, and 0.
- The collection tag and the tag list position. Each item in a collection has a tag list position as well as a collection ID. The **tag list position** of an item is the position of the item in the list of items with the same collection tag. Unlike a collection ID, the tag list positions of items with the same tag are sequential. For example, in Figure 5-4 the four items with the `'tagA'` collection tag have tag list positions 1, 2, 3, and 4. Unlike the collection ID, the tag list position of an item can change if another item with the same collection tag is added to or removed from the collection.
- The collection index. The Collection Manager assigns a **collection index** to each item in a collection. A collection index uniquely identifies its item within a collection. Indexes across a collection do not have to be sequential. The collection index of any item in a collection can change if another item is added to or removed from the collection.

Figure 5-4 Items in a collection



In Figure 5-4, the third item in the second row can be identified in three ways:

- It has a collection tag of `'tagB'` and a collection ID of 1.
- It has a collection tag of `'tagB'` and a tag list position of 3.
- It has a unique collection index assigned to it by the Collection Manager.

For examples of identifying collection items, see “Adding Items to a Collection” beginning on page 5-17, “Determining the Collection Index of an Item” beginning on page 5-19, and “Determining the Tag and ID of an Item” beginning on page 5-21.

Using the Collection Manager

This section describes how your application can

- create or dispose of a collection
- clone or copy a collection
- change the default attributes of a collection

- add items to a collection
- determine the collection index of an item
- determine the collection tag and collection ID of an item
- determine the size of an item's variable-length data
- get and set the attributes of a collection item
- replace items in a collection
- remove items from a collection
- retrieve the variable-length data from a collection item
- examine the collection tags of a collection
- flatten and unflatten collections
- read collections from and write collections to disk

Determining Whether the Collection Manager Is Available

The Collection Manager is not available in all system software versions. Therefore, before calling any Collection Manager functions, you should use the `Gestalt` function to determine whether the Collection Manager is available. To get information about the Collection Manager, you pass the `Gestalt` function the `gestaltCollectionMgrVersion` selector, as shown in Listing 5-1.

Listing 5-1 Determining whether the Collection Manager is available

```
Boolean CollectionMgrExists(long versionRequired) {

   long collectionMgrVersion;

   Boolean exists = (Gestalt(gestaltCollectionMgrVersion,
                             &collectionMgrVersion) == noErr);
   if (exists)
      exists = (collectionMgrVersion >= versionRequired);
   return(exists);
}
```

In Listing 5-1, the `CollectionMgrExists` sample function uses the `Gestalt` function to determine whether the Collection Manager is available and, if so, which version is available. If the Collection Manager is available, this sample function tests whether the version number is sufficiently high by comparing it with a specified minimum.

You would call this sample function with a line of code such as

```
exists = CollectionMgrExists(0x01008000); /* version 1.0f0 */
```

You can find out more about the `Gestalt` function in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities.*

Creating or Disposing of a Collection

The Collection Manager provides a number of ways for you to create a collection object:

- You can create a new collection object using the `NewCollection` function, as described in this section.
- You can make a copy of an existing collection object using the `CopyCollection` function, as described in "Cloning or Copying a Collection" beginning on page 5-14.
- You can create a collection from a resource using the `GetNewCollection` function, as described in "Reading Collections From and Writing Collections to Disk" beginning on page 5-41.

The `NewCollection` function creates a new collection object containing no collection items. The section "Adding Items to a Collection" beginning on page 5-17 shows how you can add items to a new collection.

The default attributes of the new, empty collection are described by the `defaultCollectionAttributes` constant, in which only the persistence attribute is set. This constant is defined in "Attributes Masks" beginning on page 5-49. The section "Changing the Default Attributes of a Collection" beginning on page 5-15 shows how you can change the default attributes of the new collection.

The owner count of the new collection is 1. You can increment the collection's owner count using the `CloneCollection` function, as shown in the section "Cloning or Copying a Collection" beginning on page 5-14. You can decrement the collection's owner count using the `DisposeCollection` function. This function decrements the owner count of a collection and frees the memory used by the collection if the owner count becomes 0.

You can find more information about the `NewCollection` function on page 5-54. You can find more information about the `DisposeCollection` function on page 5-55.

Cloning or Copying a Collection

You use the `CloneCollection` and `CopyCollection` functions to clone and copy collection objects. You clone a collection object if you want to make a copy of the reference to the collection object, and you copy a collection object if you want to make a copy of the entire object, including all of its items.

For example, if you have a reference to a collection object stored in the variable `aCollection`, you can create a new reference to this collection using

```
newCollection = CloneCollection(aCollection);
```

which increments the owner count of the collection object referenced by the `aCollection` variable and returns a copy of the reference as the function result. After

this call to the `CloneCollection` function, the `newCollection` and `aCollection` variables reference the same collection object, which has an incremented owner count.

You can create a copy of a collection object, including a copy of all its items, using

```
newCollection = CopyCollection(aCollection, nil);
```

The `CopyCollection` function does not increment the owner count of the `aCollection` collection. Instead, it creates a new collection object with an owner count of 1, copies all of the information from the `aCollection` collection into the new collection, and returns a reference to the new collection. After this call to the `CopyCollection` function, the `newCollection` and `aCollection` variables reference two distinct collections—you can make changes to one without affecting the other.

You can use the second parameter of the `CopyCollection` function to provide a reference to an existing collection object, in which case the function copies the information from the collection referenced by the first parameter into the collection referenced by the second parameter. If the collection referenced by the second parameter already has information in it, the function

- removes all of the items in the second collection—including locked items—before copying the items from the first collection into the second collection
- copies the default attribute values from the first collection into the second collection

The `CopyCollection` function does not copy the owner count or the exception procedure of the first collection; it leaves the owner count and the exception procedure of the second collection unchanged.

You can find more information about the `CloneCollection` function on page 5-56. You can find more information about the `CopyCollection` function on page 5-57.

Changing the Default Attributes of a Collection

Every collection object has default attributes. When you add a new item to a collection, the Collection Manager sets the attributes of the new item to match the default attributes of the collection. You can change the attributes of individual items in a collection using the functions described in “Getting and Setting the Attributes of an Item” beginning on page 5-24. You can change the default attributes for a collection using the `SetCollectionDefaultAttributes` function. This function allows you to change the value of a collection’s default attributes. With this function, you can change the value of every default attribute or you can choose to change only some of the default attributes.

This function takes three parameters:

- a reference to the collection object
- a mask specifying which attributes you want to edit
- the new values for the attributes

(See Figure 5-3 on page 5-10 for an overview of editing attributes.)

As an example, Listing 5-2 changes the default attributes for a collection object so that

- user attribute 0 and the lock attribute are set
- all other attributes are clear

Listing 5-2 Changing the default attributes of a collection

```
long newAttributes;
.
.
.
newAttributes = collectionUser0Mask  /* set user 0 bit */
                | collectionLockMask;  /* set lock bit */

anErr = SetCollectionDefaultAttributes(anyCollection,
                            allCollectionAttributes, /* mask */
                            newAttributes); /* new values */
```

In this example, the `allCollectionAttributes` mask, which is defined in "Attributes Masks" on page 5-49, specifies that you want to replace the value of every attribute in the collection's default attributes with the corresponding value in the `newAttributes` parameter. The value of the `newAttributes` parameter specifies that the user 0 attribute and the lock attribute are set while every other attribute is clear.

You can use different values for the second parameter of this function if you want to edit some of the collection's default attributes but leave other default attributes unchanged. For example, you could set the second parameter of this function to the `userCollectionAttributes` mask:

```
anErr = SetCollectionDefaultAttributes(anyCollection,
                            userCollectionAttributes, /* mask */
                            newAttributes); /* new values */
```

Using this mask specifies that you want to edit only the user attributes of the collection's default attributes. The function replaces the existing values for the collection's default user attributes with the values of the user attributes from the `newAttributes` parameter. In this example, the user 0 attribute is set while all the other user attributes are cleared. However, this call to the `SetCollectionDefaultAttributes` function does not change the values of any of the reserved attributes. For example, the value of the lock attribute in the collection's default attributes remains the same as it was—the value of the lock attribute in the `newAttributes` parameter makes no difference as it is not copied into the collection's default attributes.

You can find more information about the `SetCollectionDefaultAttributes` function on page 5-61.

If you want to examine the default attributes of a particular collection object, you can use the `GetCollectionDefaultAttributes` function, which is described on page 5-60.

Adding Items to a Collection

Once you've created a collection object, you can add new items to the collection using the `AddCollectionItem` function. With this function, you specify the collection tag and collection ID that you want associated with the new item, the size of the new item's variable-length data, and a pointer to the data.

Note
The Collection Manager also provides a utility function, `AddCollectionItemHdl`, which allows you to specify a handle, rather than a pointer, to the data. See page 5-92 for more information about this function. ◆

Listing 5-3 creates a new collection object and adds ten items to it. Each item has the collection tag `'GXPT'`, the items have collection IDs 0 through 9, and each item contains two coordinates that make up a QuickDraw GX point.

Listing 5-3 Adding items to a collection

```
OSErr anErr;
Collection pointsAndQuotes;
gxPoint location;
long count;
.
.
.
pointsAndQuotes = NewCollection();

location.x = ff(10);
location.y = ff(10);

for (count = 0; count < 10; count++) {
   anErr = AddCollectionItem(pointsAndQuotes,
                             'GXPT',  /* tag */
                             count,   /* id */
                             sizeof(gxPoint),     /* size */
                             &location); /* data */

   location.x += ff(1);  /* change data for next item */
   location.y += ff(1);
}
```

The collection resulting from the code in Listing 5-3 is very similar to an array: the items are numbered sequentially starting with 0, and all items are of the same size. Unlike arrays, however, collections are not limited to these properties. For example, you can add items to a collection dynamically, increasing the total number of collection items during

run time. That is, you do not have to specify the capacity of the collection at compile time. Also, collection IDs do not have to be sequential. To demonstrate, the code

```
location.x = ff(100);
location.y = ff(100);
anErr = AddCollectionItem(pointsAndQuotes,
                          'GXPT', 20, /* tag and id */
                          sizeof(gxPoint),     /* size */
                          &location); /* data */
```

adds an eleventh item to the collection from Listing 5-3. The collection tag of the new item is `'GXPT'`, but the new item has a collection ID of 20.

When you add this item to the collection, the Collection Manager assigns it a tag list position, reflecting its position in the list of items with the same collection tag. This tag list position can change if you add a new item with the same collection tag. For example, the call

```
anErr = AddCollectionItem(pointsAndQuotes,
                          'GXPT', 15, /* tag and id */
                          sizeof(gxPoint),     /* size */
                          &location); /* data */
```

adds a new item with the `'GXPT'` collection tag and a collection ID of 15. Adding this item can change the tag list position of any other item with the `'GXPT'` collection tag.

So far, the example collection contains items of the same size. You can also use the `AddCollectionItem` function to add items with variable-length data, as shown in Listing 5-4.

Listing 5-4 Adding items with variable-length data to a collection

```
anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 0,  /* tag and id */
                          17,  /* size */
                          "Le plus ca change");  /* data */

anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 1,
                          29, "Fourscore and seven years ago");

anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 2,
                          18, "It's not the heat.");
```

The sample code from Listing 5-4 adds three new items to the example collection. Each of these items, which have collection tag `'QUOT'`, contains a string of characters as its variable-length data; however, each item contains a string of different length.

Note that the `AddCollectionItem` function copies the information from the block of memory pointed to by its final parameter into the specified collection item. After adding the item, you can change your copy of the information (the copy that the last parameter points to) without affecting the value of the item's variable-length data.

You can use the `CountCollectionItems` function to count the number of items in a collection. For example, after the call

```
totalItems = CountCollectionItems(pointsAndQuotes);
```

the `totalItems` variable contains the value 15 (12 items with points and 3 items with quotes).

You can use the `CountTaggedCollectionItems` function to count the number of items in a collection that have a specified collection tag. For example, after the call

```
quotedItems = CountTaggedCollectionItems(pointsAndQuotes, 'QUOT');
```

the `quotedItems` variable contains the value 3.

For more information about the `AddCollectionItem` function, see page 5-62.

For more information about the `CountCollectionItems` and `CountTaggedCollectionItems` functions, see "Counting Items in a Collection" beginning on page 5-69.

Determining the Collection Index of an Item

Once you have added an item to a collection, you can identify that item in three ways:

- You can specify its collection tag and ID.
- You can specify its collection tag and its tag list position.
- You can specify its collection index.

A collection index is a unique value that the Collection Manager assigns to each item in a collection. You can use an item's collection index to identify the item without specifying the item's collection tag or collection ID. In fact, once you've determined the collection index of an item, specifying that item using its collection index results in faster operations than specifying the item using its collection tag and collection ID.

There are two ways to determine the collection index that the Collection Manager has assigned to an item:

- You can use the `GetCollectionItemInfo` function. With this function, you specify the collection tag and collection ID of the item, and the function returns the item's collection index.
- You can use the `GetTaggedCollectionItemInfo` function. With this function, you specify the collection tag and the tag list position of the item, and the function returns the item's collection index.

Both of these functions optionally return other information about the specified item, as shown in the next two sections.

Listing 5-5 shows how to use the the `GetCollectionItemInfo` function to determine the collection index of an item from the sample collection created in "Adding Items to a Collection" beginning on page 5-17. This listing uses the `dontWantSize` and `dontWantAttributes` constants, which are equal to `nil` and specify that you don't want to determine certain information about the item. These constants are described in "Optional Return Value Constants" on page 5-49.

Listing 5-5 Determining the index of an item

```
long index;
.
.
.
anErr = GetCollectionItemInfo(pointsAndQuotes, /* collection */
                              'GXPT', 15,  /* tag and id */
                              &index,  /* returned index */
                              dontWantSize,
                              dontWantAttributes);
```

After this call to `GetCollectionItemInfo` function, the `index` variable contains the collection index of the item in the `pointsAndQuotes` collection with the `'GXPT'` collection tag and a collection ID of 15. You can use this value to identify this item when using certain Collection Manager functions, such as `RemoveIndexedCollectionItem` and `GetIndexedCollectionItem`.

You can also use the `GetTaggedCollectionItemInfo` function to determine the collection index of a collection item. This function allows you to specify the item using the item's collection tag and tag list position. For example, in Listing 5-5, the item is specified using the `'GXPT'` collection tag and collection ID 15. Assuming the Collection Manager has assigned this item a tag list position of 11, you could also use the call

```
anErr = GetTaggedCollectionItemInfo(pointsAndQuotes,
                                    'GXPT', /* collection tag */
                                    11, /* tag list position */
                                    dontWantId,
                                    &index, /* returned index */
                                    dontWantSize,
                                    dontWantAttributes);
```

to determine the collection index for that item.

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItemInfo` and `GetTaggedCollectionItemInfo` functions, see "Getting Information About a Collection Item" beginning on page 5-76.

Determining the Tag and ID of an Item

Just as you can determine the collection index of an item given its collection tag and ID, you can also determine the collection tag and ID of an item given its collection index. Listing 5-6 shows how to determine an item's collection tag and collection ID using the `GetIndexedCollectionItemInfo` function.

Listing 5-6 Determining the tag and ID of an item given the item's index

```
long tag, id;
.
.
.
anErr = GetIndexedCollectionItemInfo(pointsAndQuotes,
                                     index, /* index of item */
                                     &tag,  /* returned tag */
                                     &id,   /* returned id */
                                     dontWantSize,
                                     dontWantAttributes);
```

You need to set the value of the `index` variable to contain the collection index of the desired item before making this call to the `GetIndexedCollectionItemInfo` function. (See the previous section, “Determining the Collection Index of an Item” on page 5-19, for in the `GetCollectionItemInfo` function shown in Listing 5-6, the `tag` variable contains the collection tag and the `id` variable contains the collection ID of the item in the `pointsAndQuotes` collection with the collection index specified by the `index` variable.

If you know the collection tag of an item and you know its tag list position, you can use the `GetTaggedCollectionItemInfo` function to determine its collection ID. For example, you could also use the call

```
anErr = GetTaggedCollectionItemInfo(pointsAndQuotes,
                                    'GXPT', 11,
                                    &id,
                                    dontWantIndex,
                                    dontWantSize,
                                    dontWantAttributes);
```

to determine the collection ID of the eleventh item in the `pointsAndQuotes` collection with the tag `'GXPT'`. With the `pointsAndQuotes` collection defined in “Adding Items to a Collection” beginning on page 5-17, the collection ID of this item turns out to be 15.

For more information about identifying collection items, see “Methods of Identifying Collection Items” on page 5-11.

For more information about the `GetCollectionItemInfo` and `GetTaggedCollectionItemInfo` functions, see “Getting Information About a Collection Item” beginning on page 5-76.

Determining the Size of an Item’s Variable-Length Data

The Collection Manager provides three functions that provide information about an item in a collection. These three functions differ in how they allow you to specify which item you want information about:

- The `GetCollectionItemInfo` function requires that you specify the collection tag and ID of the desired item.
- The `GetIndexedCollectionItemInfo` function requires that you specify the collection index of the desired item.
- The `GetTaggedCollectionItemInfo` function requires that you specify the collection tag and tag list position of the desired item.

These functions each return a variety of information about the specified item—for example, the item’s attributes, the size of the item’s variable length data, and so on. These functions return each piece of information in a separate parameter. You can specify that you do not want certain pieces of information returned by providing `nil` for the corresponding parameter. The Collection Manager provides the optional return value constants, each of which is equal to `nil`, to make your code easier to read.

Listing 5-7 shows how to use the `GetCollectionItemInfo` function to determine the size of an item's variable-length data, given that item's collection tag and ID.

Listing 5-7 Determining the size of an item's variable-length data

```
long theSize;
.
.
.
anErr = GetCollectionItemInfo(pointsAndQuotes,  /* collection */
                              'GXPT', 15,  /* tag and id */
                              dontWantIndex,
                              &theSize,  /* returned size */
                              dontWantAttributes);
```

(You can also use the `GetCollectionItemInfo` function to determine an item's collection index, as described in "Determining the Collection Index of an Item" beginning on page 5-19, or to examine an item's attributes, as described in "Getting and Setting the Attributes of an Item" beginning on page 5-24.)

Similarly, you can use the `GetIndexedCollectionItemInfo` function to determine the size of the item's variable-length data given the item's collection index:

```
anErr = GetIndexedCollectionItemInfo(pointsAndQuotes,
                                     index,  /* index of item */
                                     dontWantTag,
                                     dontWantId,
                                     &theSize, /* returned size */
                                     dontWantAttibutes);
```

(You can also use the `GetIndexedCollectionItemInfo` function to determine an item's collection tag and collection ID, as described in "Determining the Tag and ID of an Item" beginning on page 5-21, or to examine an item's attributes, as described in the next section, "Getting and Setting the Attributes of an Item.".)

Finally, you can use the `GetTaggedCollectionItemInfo` function to determine the size of the item's variable-length data given its collection tag and tag list position.

```
anErr = GetTaggedCollectionItemInfo(pointsAndQuotes,
                                    'GXPT',  /* tag of item */
                                    11,  /* tag list position */
                                    dontWantId,
                                    dontWantIndex,
                                    &theSize, /* returned size */
                                    dontWantAttributes);
```

(You can also use the `GetTaggedCollectionItemInfo` function to determine an item's collection index, as described in "Determining the Collection Index of an Item" beginning on page 5-19, to determine an item's collection ID, as described in "Determining the Tag and ID of an Item" beginning on page 5-21, or to examine an item's attributes, as described in the next section, "Getting and Setting the Attributes of an Item.")

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItemInfo`, `GetIndexedCollectionItemInfo`, and `GetTaggedCollectionItemInfo` functions, see "Getting Information About a Collection Item" beginning on page 5-76.

Getting and Setting the Attributes of an Item

The Collection Manager provides three functions that allow you to examine the attributes of a collection item:

- The `GetCollectionItemInfo` function requires that you specify the collection tag and ID of the desired item.
- The `GetIndexedCollectionItemInfo` function requires that you specify the collection index of the desired item.
- The `GetTaggedCollectionItemInfo` function requires that you specify the collection tag and tag list position of the desired item.

The Collection Manager provides two functions that allow you to edit the attributes of an item:

- The `SetCollectionItemInfo` function requires that you specify the collection tag and ID of the desired item.
- The `SetIndexedCollectionItemInfo` function requires that you specify the collection index of the desired item.

(There is no `SetTaggedCollectionItemInfo` function to correspond to the `GetTaggedCollectionItemInfo` function.)

The three information-retrieving functions allow you to determine a variety of information about the item—not just its attributes. You can find more information about the other values returned by these functions in "Determining the Collection Index of an Item" beginning on page 5-19, "Determining the Tag and ID of an Item" beginning on page 5-21, and "Determining the Size of an Item's Variable-Length Data" beginning on page 5-22.

The information-editing functions, however, allow you to edit the attributes of only the specified item. (You cannot, for instance, use these functions to change the index of an item, or the size of its variable-length data.)

Listing 5-8 shows how you can use the `GetCollectionItemInfo` function to examine the attributes of an item given the item's collection tag and collection ID. This listing uses the collection defined in "Adding Items to a Collection" beginning on page 5-17.

Listing 5-8 Examining the attributes of an item

```
long attributes;
.
.
.
anErr = GetCollectionItemInfo(pointsAndQuotes, /* collection */
                              'QUOT', 0,  /* tag and id */
                              dontWantIndex,
                              dontWantSize,
                              &attributes); /* returned attr's */
```

After this call to the `GetCollectionItemInfo` function, the `attributes` variable contains a copy of the attributes of item from the `pointsAndQuotes` collection with the collection tag `'QUOT'` and a collection ID of 0. You can examine specific attributes using the attribute bit masks, which are described in "Attribute Bit Masks" beginning on page 5-52. As an example, the expression

```
(attributes & collectionLockMask)
```

evaluates to `false` (0) if the lock attribute is not set and to `true` (not 0) if the lock attribute is set.

You can also use the `GetIndexedCollectionItemInfo` function to examine the attributes of an item, given its collection index rather than its collection tag and collection ID:

```
anErr = GetIndexedCollectionItemInfo(pointsAndQuotes,
                                     index, /* index of item */
                                     dontWantTag,
                                     dontWantId,
                                     dontWantSize,
                                     &attributes); /* returned */
```

Similarly, you can use the `GetTaggedCollectionItemInfo` function to examine the attributes of an item, given its collection tag and tag list position:

```
anErr = GetTaggedCollectionItemInfo(pointsAndQuotes,
                                    'QUOT',  /* tag of item */
                                    1,  /* tag list position */
                                    dontWantId,
                                    dontWantIndex,
                                    dontWantSize,
                                    &attributes);  /* returned */
```

You can edit the attributes of a collection item using the `SetCollectionItemInfo` and `SetIndexedCollectionItemInfo` functions. These functions require you to specify which attributes you want to edit and what the new values for those attributes should be.

You specify this information using two `long` parameters:

- The first is a mask. Each bit in this mask represents one of the item's attributes. A bit value of 0 in this mask signifies that you do not want to edit the corresponding attribute of the specified item. A bit value of 1 in this mask signifies that you do want to edit the corresponding attribute of the item.
- The second contains the new values. Each bit in this parameter represents the new value for the corresponding attribute of the item. Only the bits in this parameter that correspond to bits in the mask parameter that have a value of 1 are significant. The Collection Manager ignores the other bit values in this parameter.

Listing 5-9 shows how you can use the `SetCollectionItemInfo` to set the lock and persistence attributes of a collection item and clear all the other attributes.

Listing 5-9 Setting the lock and persistence bit attribute of an item

```
long newAttributes;
.
.
.
newAttributes = collectionLockMask
                | collectionPersistenceMask;

anErr = SetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 0, /* tag and id */
                              allCollectionAttributes, /* mask */
                              newAttributes); /* new values */
```

This example uses the `allCollectionAttributes` constant (which is defined in "Attributes Masks" beginning on page 5-49) to indicate that all the attributes of the specified collection item should be edited. As a result, the code in the example replaces the value of every attribute in the specified collection item with the corresponding value from the `newAttributes` parameter.

If you want to set the lock and persistence attributes of this collection item without affecting the values of the other attributes, you can use the `newAttributes` variable as both the mask and the values parameters:

```
anErr = SetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 0, /* tag and id */
                              newAttributes, /* mask */
                              newAttributes); /* new values */
```

In this case, the code uses the `newAttributes` parameter as the mask (which indicates that only the lock attribute and the persistence attribute should be affected) as well as the values (which indicate that both of these attributes should be set). The values of all the other attributes of the specified item remain as they were before the call.

You can also use the `SetIndexedCollectionItemInfo` function to set the attributes of an item, given the item's collection index rather than its collection tag and collection ID:

```
anErr = SetIndexedCollectionItemInfo(pointsAndQuotes,
                                     index,
                                     allCollectionAttributes,
                                     newAttributes);
```

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItemInfo`, `GetIndexedCollectionItemInfo`, and `GetTaggedCollectionItemInfo` functions, see "Getting Information About a Collection Item" beginning on page 5-76.

For more information about the `SetCollectionItemInfo` and `SetIndexedCollectionItemInfo` functions, see "Editing Item Attributes" beginning on page 5-82.

Replacing Items in a Collection

The Collection Manager provides two methods for replacing items in a collection:

- You can use the `AddCollectionItem` function, specifying the collection tag and collection ID of an existing item.
- You can use the `ReplaceIndexedCollectionItem` function, specifying the collection index of an existing item.

Note

The Collection Manager also provides the utility functions, `AddCollectionItemHdl` and `ReplaceCollectionItemHdl`, which allow you to specify a handle, rather than a pointer, to the item's data. See "Working With Macintosh Memory Manager Handles" beginning on page 5-92 for more information about these functions. ◆

The new item does not have to be the same size as the replaced item. For example, Listing 5-10 shows how you can use the `AddCollectionItem` function to replace the data in a collection item with a new data of a different length. (This example uses the collection created in "Adding Items to a Collection" beginning on page 5-17, in which the item identified by the collection tag `'QUOT'` and the collection ID 1 contains the 29-character string "Fourscore and seven years ago")

Listing 5-10 Replacing an item in a collection

```
anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 1,
                          22, "Eighty-seven years ago");
```

You cannot replace a collection item if its lock attribute is set. For example, the previous section shows how to set the lock attribute of the item with the collection tag `'QUOT'` and the collection ID 0. If you try to replace this item using

```
anErr = AddCollectionItem(pointsAndQuotes,
                          'QUOT', 0,
                          24, "Plus c'est la meme chose");
```

the code sets the `anErr` variable to the `collectionItemLockedErr` value and the Collection Manager does not replace the item.

If you know the collection index of an item, you can use the `ReplaceIndexedCollectionItem` function to replace the item. This function finds the specified item more efficiently than the `AddCollectionItem` function. Listing 5-11 shows an example of this function.

Listing 5-11 Replacing an item using the item's index

```
long index;
.
.
.
/* find the index. */
anErr = GetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 1,
                              &index,
                              dontWantSize,
                              dontWantAttributes);
.
.
.
/* replace the item. */
anErr = ReplaceIndexedCollectionItem(pointsAndQuotes,
                                     index,
                                     22,
                                     "Eighty-seven years ago");
```

The example in Listing 5-11 uses the `GetCollectionItemInfo` function to find the collection index that the Collection Manager has assigned to the item with a collection tag of `'QUOT'` and a collection ID of 1. Finding this collection index requires some processing time. However, once you've found the item's collection index, you can use it to find information about the item quickly, because functions that search for a collection item using the item's collection index operate more efficiently than functions that search using the item's collection tag and collection ID. Typically, if you want to search for an item only once, you use the item's collection tag and collection ID. If you know that you have to search for the same item repeatedly, you find the item's collection index and use the collection index when examining or editing the item.

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `AddCollectionItem` function and the `ReplaceIndexedCollectionItem` function, see "Adding and Replacing Items in a Collection" beginning on page 5-62.

Removing Items From a Collection

The Collection Manager provides two methods for removing individual items from a collection:

- You can use the `RemoveCollectionItem` function, specifying the collection tag and collection ID of the item you want to remove.
- You can use the `RemoveIndexedCollectionItem` function, specifying the collection index of the item you want to remove.

The Collection Manager provides three methods for removing multiple items from a collection:

- You can use the `PurgeCollection` function to remove all the items in a collection whose attributes match criteria you specify.
- You can use the `PurgeCollectionTag` function to remove all the items in a collection that have a specified collection tag.
- You can use the `EmptyCollection` function to remove every item from a collection.

Listing 5-12 shows how you can use the `RemoveCollectionItem` function to remove an item from a collection. (This example uses the collection created in "Adding Items to a Collection" beginning on page 5-17.)

Listing 5-12 Removing an item in a collection

```
anErr = RemoveCollectionItem(pointsAndQuotes,
                             'QUOT', 1); /* tag and id */
```

You can remove a collection item even if its lock attribute is set—the lock attribute only affects replacing. For example, if you have set the lock attribute of the collection item with the collection tag `'QUOT'` and the collection ID 0, you can remove this item using

```
anErr = RemoveCollectionItem(pointsAndQuotes,
                             'QUOT', 0); /* tag and id */
```

You can also remove the item using

```
anErr = RemoveCollectionItem(pointsAndQuotes,
                             'QUOT', 0); /* tag and id */
```

If you know the index of an item, you can use the `RemoveIndexedCollectionItem` function to remove the item. This function finds the specified item more efficiently than the `RemoveCollectionItem` function. Listing 5-13 shows an example of this function.

Listing 5-13 Removing an item using the item's index

```
long index;
.
.
.
/* get the index */
anErr = GetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 1,
                              &index,
                              dontWantSize,
                              dontWantAttributes);
.
.
.
/* remove the item */
anErr = RemoveIndexedCollectionItem(pointsAndQuotes, index);
```

The example in Listing 5-13 uses the `GetCollectionItemInfo` function to find the collection index that the Collection Manager has assigned to the item with a collection tag of `'QUOT'` and a collection ID of 1. Finding this collection index requires some processing time. However, once you've found the item's collection index, you can use it to find information about the item quickly, because functions that search for a collection item using the item's collection index operate more efficiently than functions that search using the item's collection tag and collection ID.

The `PurgeCollection` function allows you to remove multiple items from a collection. You provide this function with a collection and a set of attribute values, and it removes any items in the collection whose attributes match these values. You specify which attributes to examine in the second parameter of this function, and you specify the values to compare those attributes against in the third parameter, as shown in Listing 5-14.

Listing 5-14 Removing multiple items with specific attributes

```
long whichAttributes, attributeValues;
.
.
.
/* specify which attributes to examine: user 0 and user 1 */
whichAttributes = collectionUser0Mask
                  | collectionUser1Mask;
```

```
/* specify the values to test for: user 0 set and user 1 clear */
attributeValues = collectionUser0Mask
                  & ~collectionUser1Mask;

/* purge all items with user 0 attribute set and user 1 clear */
PurgeCollection(pointsAndQuotes,
                whichAttributes,
                attributeValues);
```

This example sets two bits in the `whichAttributes` variable—the user 0 attribute and the user 1 attribute—and clears every other bit in this variable, which signifies that the function should test only the user 0 attribute and the user 1 attribute. The `attributeValues` variable sets the user attribute 0 flag and clears the user attribute 1 flag. Therefore, this call to `PurgeCollection` removes every item in the collection that has the user 0 attribute set and the user 1 attribute clear. It ignores the values of all the other attributes.

You can use the `PurgeCollectionTag` function to remove all of the items in a collection that share a collection tag—even the locked items. To remove all the items with the collection tag `'GXPT'` from the `pointsAndQuotes` collection (which is defined in “Adding Items to a Collection” beginning on page 5-17), you could use this line of code:

```
PurgeCollectionTag(pointsAndQuotes, 'GXPT');
```

Finally, you can remove all of the items in a collection—even the locked items—using the `EmptyCollection` function:

```
EmptyCollection(pointsAndQuotes);
```

For more information about identifying collection items, see “Methods of Identifying Collection Items” on page 5-11.

For more information about the `RemoveCollectionItem`, `RemoveIndexedCollectionItem`, `PurgeCollection`, `PurgeCollectionTag`, and `EmptyCollection` functions, see “Removing Items From a Collection” beginning on page 5-65.

Retrieving the Variable-Length Data From an Item

The Collection Manager provides three functions that return a copy of the information in an item's variable-length data. These three functions differ in how they allow you to specify which item you want information about:

- The `GetCollectionItem` function requires that you specify the collection tag and collection ID of the desired item.
- The `GetIndexedCollectionItem` function requires that you specify the collection index of the desired item.
- The `GetTaggedCollectionItem` function requires that you specify the collection tag and tag list position of the desired item.

Note
The Collection Manager also provides the utility function `GetCollectionItemHdl`, which returns a copy of the item's data in a block of memory referenced by a Macintosh Memory Manager handle, rather than a pointer. See page 5-94 for more information about this function. ◆

These functions each return two pieces of information about the specified item—the size of its variable-length data and a copy of the data itself. You can specify that you want to determine either the size or the data or both (or neither, actually, although that doesn't prove to be very useful).

Typically, you call these functions twice:

- once to determine the size of the data (if you don't already know the size)
- once (after allocating enough memory) to obtain a copy of the data.

Listing 5-15 shows how to use the `GetCollectionItem` function to retrieve the variable-length data from an item. This sample code uses the `pointsAndQuotes` collection defined in "Adding Items to a Collection" beginning on page 5-17.

Listing 5-15 Retrieving the variable-length data from an item

```
long theSize;
char *theData;
.
.
.
anErr = GetCollectionItem(pointsAndQuotes,
                          'QUOT', 0, /* tag and id */
                          &theSize,
                          dontWantData);

theData = (char *) NewPtr(theSize);
```

```
anErr = GetCollectionItem(pointsAndQuotes,
                          'QUOT', 0,
                          dontWantSize,
                          theData);
```

If you specify a non-`NIL` value for the size parameter, the `GetCollectionItem` function returns in the size parameter the actual number of bytes of the item's data.

If you specify non-`NIL` values for both the size and data parameters, the number of bytes returned in the data parameter is either the value specified by the size parameter or the actual number of bytes of the specified item's data, whichever is lower.

You can also use the `GetIndexedCollectionItem` function to retrieve an item's data, given the item's collection index rather than its collection tag and collection ID, as shown in Listing 5-16.

Listing 5-16 Retrieving the variable-length data from an item using the item's index

```
long index;
long theSize;
char *theData;
.
.
.
/* get the index and data size */
anErr = GetCollectionItemInfo(pointsAndQuotes,
                              'QUOT', 0, /* tag and id */
                              &index,
                              &theSize,
                              dontWantAttributes;
.
.
.
theData = (char *) NewPtr(theSize);

anErr = GetIndexedCollectionItem(pointsAndQuotes,
                                 index,
                                 dontWantSize,
                                 theData);
```

Similarly, you can use the `GetTaggedCollectionItem` function to retrieve an item's data, given the item's collection tag and tag list position, as shown in Listing 5-17.

Listing 5-17 Retrieving the variable-length data from an item using the tag and tag list position

```
long index;
long theSize;
char *theData;
.
.
.
anErr = GetTaggedCollectionItem(pointsAndQuotes,
                                'QUOT',
                                1, /* first of the 'QUOT' items */
                                &theSize,
                                dontWantData);

theData = (char *) NewPtr(theSize);

anErr = GetTaggedCollectionItem(pointsAndQuotes,
                                'QUOT',
                                1, /* first of the 'QUOT' items */
                                dontWantSize,
                                (void *) theData);
```

For more information about identifying collection items, see "Methods of Identifying Collection Items" on page 5-11.

For more information about the `GetCollectionItem`, `GetIndexedCollectionItem`, and `GetTaggedCollectionItem` functions, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-70.

Examining the Collection Tags of a Collection

The Collection Manager provides three functions that allow you to examine the collection tags contained in a specific collection:

- You can use the `CollectionTagExists` function to determine if any of the items in a specific collection have a specified collection tag.
- You can use the `CountCollectionTags` function to determine the total number of distinct collection tags contained in the items of a collection.
- You can use the `GetIndexedCollectionTag` function to examine the value of one of the distinct collection tags contained in a collection.

Every collection has a list of distinct collection tags contained in that collection. The `GetIndexedCollectionTag` function allows you to step through this list of distinct collection tags, as shown in Listing 5-18.

Listing 5-18 Counting tags in a collection

```
long numTags, numItems, eachTag, eachItem;
.
.
.
numTags = CountCollectionTags(pointsAndQuotes);

/* iterate through each tag */
for (eachTag = 1; eachTag <= numTags; ++eachTag) {
   GetIndexedCollectionTag(pointsAndQuotes, eachTag, &theTag);
   numItems = CountTaggedCollectionItems(pointsAndQuotes, theTag);

   /* iterate through each item with that tag */
   for (eachItem = 1; eachItem <= numItems; ++eachItem) {

      /* find size of item data and obtain copy of data */
      GetTaggedCollectionItem(pointsAndQuotes,
                              theTag, eachItem,
                              &theSize, dontWantData);
      theData = (char *) NewPtr(theSize);
      GetTaggedCollectionItem(pointsAndQuotes,
                              theTag, eachItem,
                              dontWantSize, theData);
      .
      .
      .
      /* manipulate item data . . .*/
      .
      .
      .
      DisposePtr(theData);
   }
}
```

This sample code determines the total number of distinct tags in the `pointsAndQuotes` collection using the `CountCollectionTags` function. Then, it uses the `GetIndexedCollectionTag` function to step through each of the distinct collection tags in the collection.

With each collection tag, the sample code uses the `GetTaggedCollectionItem` function to retrieve the variable-length data from each item with the tag. In this manner, this sample code retrieves the data from every item in the collection.

For more information about the `CollectionTagExists`, `CountCollectionTags`, and `GetIndexedCollectionTag` functions, see "Getting Information About Collection Tags" beginning on page 5-85.

Flattening and Unflattening a Collection

The Collection Manager provides the `FlattenCollection` function for converting the information in a collection object into a flattened stream of bytes. With the `FlattenCollection` function, you provide a callback function that operates on the stream of bytes—you can use this callback function to write the stream out to disk, store the stream in a Macintosh Memory Manager handle, and so on.

The `FlattenCollection` function takes three parameters:

- a reference to the collection to flatten
- a pointer to the callback function that you provide to handle the returned stream of bytes
- a 32-bit reference constant that the Collection Manager passes back to your callback function

When you call the `FlattenCollection` function, the Collection Manager begins converting the collection into a stream of bytes. It repeatedly calls your callback function, each time sending it more of the flattened collection, until it has converted the entire collection.

Your callback function determines what happens to the flattened collection. This function must take three parameters: a `long` value that represents the size of the current block of data, a pointer to the current block of data, and a reference constant that you can use as a pointer to other information.

Listing 5-19 shows an example callback function. This function appends the block of data provided by the Collection Manager in the `theData` parameter to the end of a block of data referenced by a Macintosh Memory Manager handle. The handle and the current size of the block of data referenced by the handle are stored in a `TFlattenBlock` structure. (The sample code in Listing 5-20 passes a pointer to this structure as the reference constant when calling the `FlattenCollection` function, which passes the pointer back to your callback function.)

Listing 5-19 Flattening procedure

```
typedef struct {
   long position;
   Handle dataHandle;
} TFlattenBlock;

OSErr FlattenProc(long theSize, Ptr theData,
                  TFlattenBlock *flattenBlock) {

   register OSErr anErr = noErr;


   SetHandleSize(flattenBlock->dataHandle,
                    flattenBlock->position + theSize);
   anErr = MemError();
   if (anErr == noErr) {
      BlockMove(data,
                *flattenBlock->dataHandle +
                flattenBlock->position,
                theSize);
         flattenBlock->position += theSize;
      }
   }
   return anErr;
}
```

Listing 5-20 shows how you can use this callback function. The sample function in Listing 5-20 uses the `FlattenCollection` function to flatten a collection into a block of memory referenced by a Macintosh Memory Manager handle.

Listing 5-20 The `FlattenCollectionToHdl` function

```
/* possible implementation of FlattenCollectionToHdl */

OSErr FlattenCollectionToHdl(Collection anyCollection,
                             Handle flattenedCollection)
{

   register OSErr anErr;
   TFlattenBlock flattenBlock;

   flattenBlock.position = 0;
   flattenBlock.dataHandle = flattenedCollection;

   if (!(anErr = MemError())) {

      anErr = FlattenCollection(anyCollection,
                                FlattenProc,
                                &flattenBlock);

      if (anErr)
         flattenBlock.dataHandle = nil;
   }

   return anErr;
}
```

This function creates a `TFlattenBlock` structure, initializes the `position` field to 0, and initializes the `dataHandle` field to a newly allocated Macintosh Memory Manager handle. The function then calls the `FlattenCollection` function, specifying the collection to flatten, the callback function specified in Listing 5-19, and a pointer to the `TFlattenBlock` structure. In response, the Collection Manager flattens the specified collection one piece at a time, repeatedly calling the callback function with new blocks of the flattened collection. The Collection Manager provides a pointer to the `TFlattenBlock` structure when calling the callback function. The callback function uses this information to copy each new block of flattened collection data onto the end of the Macintosh Memory Manager handle.

Listing 5-21 shows the reverse process—using the `UnflattenCollection` function to convert a flattened collection from a Macintosh Memory Manager handle into a collection object.

Listing 5-21 A possible implementation of the `UnflattenCollectionFromHdl` function

```
void UnflattenProc(long theSize, Ptr theData,
                   TFlattenBlock *flattenBlock) {

   BlockMove(*flattenBlock->dataHandle +
             flattenBlock->position,
             theData, theSize)

   flattenBlock->position += theSize;
}


OSErr UnflattenCollectionFromHdl(Collection anyCollection,
                                 Handle flattenedCollection)
{
   register OSErr anErr;
   TFlattenBlock flattenBlock;

   flattenBlock.position = 0;
   flattenBlock.dataHandle = flattenedCollection;

   anErr = UnflattenCollection(anyCollection,
                               UnflattenProc,
                               &flattenBlock);

   return anErr;
}
```

Listing 5-21 shows a possible implementation of the `UnflattenCollectionFromHdl` function. The Collection Manager provides both the `FlattenCollectionToHdl` and `UnflattenCollectionFromHdl` functions for you—you do not have to define these yourself. For more information about the flattening and unflattening functions, see “Flattening and Unflattening a Collection” beginning on page 5-88.

Reading Collections From and Writing Collections to Disk

The Collection Manager provides a number of methods for storing collections on disk:

- You can store the collection's contents as a collection (`'cltn'`) resource and read the information into a collection object using the `GetNewCollection` function. For more information about the `'cltn'` resource, see "The Collection Resource" beginning on page 5-102, and for more information about the `GetNewCollection` function, see the description of that function on page 5-99. For an example of reading a collection object from a collection resource, see the next section, "Reading a Collection From a Collection Resource."
- You can flatten a collection using the `FlattenCollection` function and provide a callback function that writes the blocks of flattened data to a file. You can unflatten this collection using the `UnflattenCollection` function, providing a callback function that reads blocks of data from the file. For more information about these functions, see "Flattening and Unflattening a Collection" beginning on page 5-37 and the description of the `FlattenCollection` function on page 5-88 and the description of the `UnflattenCollection` function on page 5-90.
- You can flatten a collection to a handle using the `FlattenCollectionToHdl` function and write the contents of the handle to the resource fork of a file (using the Macintosh function `AddResource`) or to the data fork of a file (using the Macintosh function `FSWrite`). You can then read the contents of this file into a handle (using the Macintosh functions `GetResource` or `FSRead`) and unflatten the result using the `UnflattenCollectionFromHdl` function.

IMPORTANT

Although you may create a resource containing a flattened collection using the `FlattenCollectionToHdl` and `AddResource` functions, you cannot recreate the collection from this resource using the `GetNewCollection` function. The resource format created by the `FlattenCollectionToHdl` and `AddResource` functions is incompatible with the resource format expected by the `GetNewCollection` function. ▲

Listing 5-22 shows how to flatten a collection to a handle and then write the contents of the handle to the resource fork of a disk file.

Listing 5-22 Flattening a collection to a disk file as a resource

```
OSErr anErr;
Handle flattened;

/* write the collection out as a resource */

flattened = NewHandle(0);
anErr = FlattenCollectionToHdl(myCollection, flattened);

if (anErr == noErr) {
   AddResource(flattened, myType, myID, myName);
   anErr = ResError();
}
```

Listing 5-23 shows how to flatten a collection to a handle and then write the contents of the handle to the data fork of a disk file.

Listing 5-23 Flattening a collection to a data fork of a disk file

```
OSErr anErr;
Handle flattened;
long theSize;

/* write the collection out to the data fork */

flattened = NewHandle(0);
anErr = FlattenCollectionToHdl(myCollection, flattened);

if (anErr == noErr) {
   theSize = GetHandleSize(flattened);
   anErr = FSWrite(refNum, theSize, *flattened);
}
```

Listing 5-24 shows how to read a flattened collection from the resource fork of a disk file into a block of memory referenced by a Macintosh Memory Manager handle and then unflatten the information in that block of memory into a collection object.

Listing 5-24 Unflattening a collection from a disk file as a resource

```
Handle flattened;
Collection myCollection;

if (myCollection = NewCollection()) {

   /* read the collection in as a resource */

   flattened = GetResource(myType, myID);

   if ((anErr = ResError()) == noErr) {
   anErr = UnflattenCollectionFromHdl(myCollection, flattened);
   ReleaseResource(flattened);
   if (anErr == noErr)
      anErr = ResError();
   }
}
```

Listing 5-25 shows how to read a flattened collection from the data fork of a disk file into a block of memory referenced by a Macintosh Memory Manager handle and then unflatten the information in that block of memory into a collection object.

Listing 5-25 Unflattening a collection from the data fork of a disk file

```
OSErr anErr;
Handle flattened;
Collection myCollection;

if (myCollection = NewCollection()) {

   /* read the collection in from the data fork */

   flattened = NewHandle(theSize);

   if ((anErr = MemError()) == noErr) {
```

```
      if ((anErr = FSRead(refNum, theSize, *flattened)) == noErr)
         anErr = UnflattenCollectionFromHdl(myCollection,
                                            flattened);
      DisposHandle(flattened);

   }
}
```

To unflatten a collection using Listing 5-25, you must know the size of the collection before you can unflatten it. If you don't know the size of the collection, you unflatten a collection using the callback function mechanism described in "Flattening and Unflattening a Collection" beginning on page 5-37.

For more information about the `FlattenCollectionToHdl` function and the `UnflattenCollectionFromHdl` function, see "Flattening and Unflattening a Collection" beginning on page 5-37 as well as the descriptions of these functions starting on page 5-97.

For information about the Macintosh functions `AddResource` and `GetResource`, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. For information about the Macintosh functions `FSRead` and `FSWrite`, see the chapter "File Manager" in *Inside Macintosh: Files*.

Reading a Collection From a Collection Resource

To store a collection to disk, you can flatten a collection and write the flattened data to a file, as described in the previous section, or you can create a **collection (`'cltn'`) resource.** The format of the collection resource is shown in the section "The Collection Resource" beginning on page 5-102.

You can create a collection object from the information stored in a collection resource using the `GetNewCollection` function. Listing 5-26 gives an example.

Listing 5-26 Reading a collection from a collection resource

```
OSErr ReadCollectionFromResource(short refNum, short resID,
                                 Collection* pCollection)
{
   OSErr anErr = noErr;

   short saveResFile = CurResFile();
   UseResFile(refNum);
   *pCollection = GetNewCollection(resID);
```

```
   if (!*pCollection) {
      anErr = ResError();
      if (!anErr)             /* if ResErr returned noErr */
         anErr = resNotFound; /* then the error was resNotFound */
   }

   UseResFile(saveResFile);

   return anErr;
}
```

The `ReadCollectionFromResource` sample function requires three parameters:

- the reference number of the file containing the desired collection resource
- the resource ID of the desired collection resource
- a pointer to a collection object reference

The sample function uses the `CurResFile` function to determine the current resource file, saves the reference number of that resource file, and uses the `UseResFile` function to indicate that the current resource file should be the resource file specified by the reference number contained in the first parameter.

The sample function then uses the `GetNewCollection` function, which takes a resource ID as its only parameter, to read the information from the designated collection resource into the collection object referenced by the sample function's third parameter.

Finally, the sample function checks for errors and resets the current resource file.

For more information about resource files and the `CurResFile`, `UseResFile`, and `ResError` functions, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

For more information about the collection resource, see "The Collection Resource" beginning on page 5-102. For more information about the `GetNewCollection` function, see the description of that function on page 5-99.

Installing an Exception Procedure

The Collection Manager allows you to specify an exception procedure for each collection object. When you attempt to manipulate a collection object using a Collection Manager function and the function results in an error, the Collection Manager calls the exception procedure for the collection object and sends it two parameters: a reference to the collection object that caused the error and the error code that was generated.

In an exception procedure, you can handle the error and then change the error code to `noErr`, a process which indicates that the Collection Manager can return control to the place in your application that generated the error as if no error had occurred. You can also change the error from one error code to another. A third alternative is to use the

ANSI C functions `setjmp` and `longjmp` to jump out of the exception handler and into code to handle the error. Listing 5-27 shows a sample exception procedure.

Listing 5-27 A sample exception procedure

```
jmp_buf cpuState; /* global machine state */

pascal OSErr MyExceptionHandler(Collection errorCollection,
                                OSErr status)
{
   /* ignore collectionItemLockedErr errors */
   if (status == collectionItemLockedErr)
      return noErr;

   /* all other errors must be handled by caller's setjmp block */
   /* jump back to callers setjmp block and return status */
   longjmp(cpuState, status);
}

void ExceptionTest(Collection anyCollection)
{
   OSErr result;

   SetCollectionExceptionProc(anyCollection, MyExceptionHandler);

   if (!(result = setjmp(cpuState))) {
      AddCollectionItem(anyCollection, 'tag1', 1, 4, "data");
      AddCollectionItem(anyCollection, 'tag1', 2, 9, "more data");
      AddCollectionItem(anyCollection, 'tag1', 3, 9, "last data");

      /* cause an error . . . */
      RemoveCollectionItem(anyCollection, 'tag1', 4);
   } else {
      .
      .
      .
      /* handle errors other than collectionItemLockedErr */
      /* use result local variable to determine which error */
      .
      .
      .
   }
}
```

In Listing 5-27, the `ExceptionTest` sample function takes a single parameter: a reference to a collection object. The sample function first calls the `SetCollectionExceptionProc` function to install an exception handler for this collection object. In this example, the call to `SetCollectionExceptionProc` installs the `MyExceptionHandler` function as the exception handler.

The next line of the `ExceptionTest` sample function calls the `setjmp` function. This function stores the current machine state, including the current position in the sample code, into the `cpuState` global variable. It also returns a value of 0 as its function result, and this value is assigned to the local variable `result`. This value is negated (by the `!` operator), an operation that produces a Boolean value of `true`. Therefore, the block of code in the `if` clause begins to execute.

Imagine that the first call to the `AddCollectionItem` function completes successfully, but that the second call to `AddCollectionItem` generates a `collectionItemLockedErr` error. During the second call to `AddCollectionItem`, the Collection Manager responds to the error by calling the `MyExceptionHandler` function. The first parameter passed to this function indicates the collection that generated the error, and the second parameter passed to this function indicates the error that was generated. This sample exception handler determines whether the error is the `collectionItemLockedErr` error (which it is in this example) and then returns with the `noErr` error as the function result. The Collection Manager notices this change in error and returns control to the sample function as if no error had occurred. (Just as you can use this mechanism to ignore certain errors, you can also use this mechanism to change errors of one type into errors of another type.) Since effectively no error has now occurred, the `ExceptionTest` sample function continues by executing the third call to the `AddCollectionItem` function.

The subsequent line of the `ExceptionTest` sample function attempts to remove an item that is not in the collection, resulting in a `collectionItemNotFoundErr` error. Again, the Collection Manager responds by calling the exception handler. In this case, however, the error is not the `collectionItemLockedErr` error, so the exception handler executes this line of code:

```
longjmp(cpuState, status);
```

When you call the `longjmp` function,

- control is passed to the location of the corresponding call to the `setjmp` function
- the value passed as the second parameter to the `longjmp` function becomes the function result of the `setjmp` function

Therefore, this call to the `longjmp` function passes control back to the location in the `ExceptionTest` sample function where `setjmp(cpuState)` was called earlier. This time, however, the function result returned by the `setjmp` function is not 0, as it was before, but instead is the value of `status`, the second parameter in the call to the `longjmp` function. Therefore, the function result of the `setjmp` function is set to be the `collectionItemNotFoundErr` error.

Once again, the `ExceptionTest` sample function assigns this function result to the `result` local variable, and negates it with the `!` operator. This time the negation produces a Boolean value of `false`, and therefore the block of code in the `else` clause begins to execute. In this block of code, you can handle errors not handled in the exception handler, using the `result` local variable to determine which error occurred.

You can find more information about the `SetCollectionExceptionProc` function on page 5-59. You can find more information about exception procedures on page 5-101.

Collection Manager Reference

This section provides reference information about the data types, functions, and resources that allow you to create and manipulate collection objects. It includes

- type definitions of the data types, including enumeration types, that are specific to the Collection Manager
- descriptions of the functions that operate on collection objects and their items
- descriptions of the application-defined callback function used for flattening and unflattening collections and the application-defined callback function used for exception handling
- the definition of the resource type used to store collection objects on disk

Data Types

This section describes the data types that you use to obtain information from and provide information to the Collection Manager functions.

Collection Objects

The Collection Manager provides you with access to a collection object through a `Collection` reference:

```
typedef struct PrivateCollectionRecord *Collection;
```

The `Collection` type defines a reference type that your compiler can type-check; it does not define a pointer to a publicly defined data structure. The contents of the collection object are private; you must use the Collection Manager functions to manipulate collection objects.

Collection Tags

Each item in a collection is uniquely identified by its collection tag and its collection ID. The collection tag is a four-character identifier, similar to the indentifiers used for resources:

```
typedef long CollectionTag;   /* 4-byte identifier ('xxxx') */
```

For more information about collection tags, see “Collection Items” beginning on page 5-8.

Optional Return Value Constants

Many of the Collection Manager functions return multiple pieces of information. For most of these functions, you can specify that you don’t want a specific piece of information to be returned by specifying `nil` for the corresponding parameter when calling the function.

The Collection Manager provides the optional return value constants to make your code easier to read when specifying that you are not interested in obtaining certain types of information:

```
enum {
   dontWantTag = 0L,
   dontWantId = 0L,
   dontWantSize = 0L,
   dontWantAttributes = 0L,
   dontWantIndex = 0L,
   dontWantData = 0L
};
```

You can use these enumeration constants in place of the more generic constant `nil` when specifying that you don’t want to receive certain optional return values from a function.

Attributes Masks

The Collection Manager provides four convenient attributes masks that you can use when specifying attributes for any of the attribute-related Collection Manager functions:

```
enum {
   noCollectionAttributes = 0x00000000,
   allCollectionAttributes = 0xFFFFFFFF,
   userCollectionAttributes = 0x0000FFFF,
   defaultCollectionAttributes = 0x40000000
};
```

Constant descriptions

`noCollectionAttributes`
: Specifies a mask in which all collection attributes are clear. You might use this constant when clearing all the attributes of an item or when testing whether all of an item's attributes are clear.

`allCollectionAttributes`
: Specifies a mask in which all collection attributes are set. You might use this constant as a mask to indicate that you want to edit or test every attribute of an item, or you might use it to set every attribute of an item.

`userCollectionAttributes`
: Specifies a mask in which the user attributes are set and the reserved attributes are clear. You might use this constant as a mask to indicate that you want to edit or test only the user attributes of an item, or you might use it to set every user attribute of an item.

`defaultCollectionAttributes`
: Specifies a mask in which the persistent attribute is set and all other attributes are clear. You might use this constant when testing to see if an item's attributes have been edited.

You can also use the attribute bit masks, described on page 5-52, as masks for individual attributes.

For more information about collection item attributes, see "Collection Items" beginning on page 5-8.-

Attribute Bit Numbers

The Collection Manager provides the attribute bit numbers enumeration to provide constant names for each of the bits in a collection item's attributes.

```
enum {
   collectionUser0Bit = 0,  /* for use by your application */
   collectionUser1Bit = 1,
   collectionUser2Bit = 2,
   collectionUser3Bit = 3,
   collectionUser4Bit = 4,
   collectionUser5Bit = 5,
   collectionUser6Bit = 6,
   collectionUser7Bit = 7,
   collectionUser8Bit = 8,
   collectionUser9Bit = 9,
   collectionUser10Bit = 10,
   collectionUser11Bit = 11,
   collectionUser12Bit = 12,
```

```
    collectionUser13Bit = 13,
    collectionUser14Bit = 14,
    collectionUser15Bit = 15,

    collectionReserved0Bit = 16,  /* reserved for use by Apple */
    collectionReserved1Bit = 17,
    collectionReserved2Bit = 18,
    collectionReserved3Bit = 19,
    collectionReserved4Bit = 20,
    collectionReserved5Bit = 21,
    collectionReserved6Bit = 22,
    collectionReserved7Bit = 23,
    collectionReserved8Bit = 24,
    collectionReserved9Bit = 25,
    collectionReserved10Bit = 26,
    collectionReserved11Bit = 27,
    collectionReserved12Bit = 28,
    collectionReserved13Bit = 29,

    collectionPersistenceBit = 30,  /* Currently defined */
    collectionLockBit = 31
};
```

The lower 16 bits of the attributes property of a collection item represent the user-defined attributes. You can use these attributes for any purpose suitable to your application.

The upper 16 bits are reserved for use by Apple Computer, Inc. Currently, the 2 high bits are defined: bit 30 represents the persistence attribute and bit 31 represents the lock attribute.

For more information about collection item attributes, see “Collection Items” beginning on page 5-8.

Attribute Bit Masks

Using the attribute bit numbers, the Collection Manager provides convenient attribute masks for each of the attributes:

```
enum {
   collectionUser0Mask = 1L << collectionUser0Bit,
   collectionUser1Mask = 1L << collectionUser1Bit,
   collectionUser2Mask = 1L << collectionUser2Bit,
   collectionUser3Mask = 1L << collectionUser3Bit,
   collectionUser4Mask = 1L << collectionUser4Bit,
   collectionUser5Mask = 1L << collectionUser5Bit,
   collectionUser6Mask = 1L << collectionUser6Bit,
   collectionUser7Mask = 1L << collectionUser7Bit,
   collectionUser8Mask = 1L << collectionUser8Bit,
   collectionUser9Mask = 1L << collectionUser9Bit,
   collectionUser10Mask = 1L << collectionUser10Bit,
   collectionUser11Mask = 1L << collectionUser11Bit,
   collectionUser12Mask = 1L << collectionUser12Bit,
   collectionUser13Mask = 1L << collectionUser13Bit,
   collectionUser14Mask = 1L << collectionUser14Bit,
   collectionUser15Mask = 1L << collectionUser15Bit,

   collectionReserved0Mask = 1L << collectionReserved0Bit,
   collectionReserved1Mask = 1L << collectionReserved1Bit,
   collectionReserved2Mask = 1L << collectionReserved2Bit,
   collectionReserved3Mask = 1L << collectionReserved3Bit,
   collectionReserved4Mask = 1L << collectionReserved4Bit,
   collectionReserved5Mask = 1L << collectionReserved5Bit,
   collectionReserved6Mask = 1L << collectionReserved6Bit,
   collectionReserved7Mask = 1L << collectionReserved7Bit,
   collectionReserved8Mask = 1L << collectionReserved8Bit,
   collectionReserved9Mask = 1L << collectionReserved9Bit,
   collectionReserved10Mask = 1L << collectionReserved10Bit,
   collectionReserved11Mask = 1L << collectionReserved11Bit,
   collectionReserved12Mask = 1L << collectionReserved12Bit,
   collectionReserved13Mask = 1L << collectionReserved13Bit,

   collectionPersistenceMask = 1L << collectionPersistenceBit,
   collectionLockMask = 1L << collectionLockBit
};
```

You can use these attribute masks when testing or setting a particular collection item attribute.

For more information about collection attributes, see “Collection Attributes” beginning on page 5-9.

For an example using these attributes, see “Getting and Setting the Attributes of an Item” beginning on page 5-24.

Functions

This section describes the Collection Manager functions you can use to

- create and dispose of collection objects
- clone and copy collection objects and determine their owner counts
- get and set the default attributes for a collection object
- add and replace items in a collection
- remove items from a collection
- count items in a collection
- retrieve the variable-length data from a collection item
- get information about an item in a collection (for example, the index of the item, the size of the item’s data, or the item’s attribute flags)
- set the attribute flags of a collection item
- get information about the collection tags associated with the items of a collection
- flatten and unflatten collections
- use Macintosh Memory Manager handles to specify variable-length data

▲ **WARNING**
Many of the functions in this section require a reference to a collection object (that is, a reference of type `Collection`) as a parameter. When calling any of these functions, you must always provide a valid collection object reference. If you do not, the behavior of the function is undefined. ▲

Creating and Disposing of Collection Objects

The functions described in this section allow you to work with collections as objects in memory. With the functions in this section, you can create new, empty collection objects and dispose of existing collection objects.

You use the `NewCollection` function to create a new collection object and the `DisposeCollection` function to dispose of a collection object.

NewCollection

You can use the `NewCollection` function to create a new, empty collection object.

```
Collection NewCollection(void);
```

function result A reference to the newly created collection object.

DESCRIPTION

The `NewCollection` function allocates memory for a new collection object, initializes it, and returns a reference to it as the function result. The new collection contains no items and has an owner count of 1.

The `NewCollection` function does not return an error code; it returns `nil` if it cannot create a new collection object.

SPECIAL CONSIDERATIONS

You are responsible for disposing of collection objects that you create with this function when you no longer need them. See the next section, which describes the `DisposeCollection` function, for information about disposing of collection objects.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter "Introduction to QuickDraw GX" in *Inside Macintosh: QuickDraw GX Objects*.

For examples using this function, see "Creating or Disposing of a Collection" beginning on page 5-14 and "Adding Items to a Collection" beginning on page 5-17.

To create a copy of an existing collection object, use the `CopyCollection` function, which is described in the previous section.

To dispose of a collection object, use the `DisposeCollection` function, which is described in the next section.

DisposeCollection

You can use the `DisposeCollection` function to dispose of a collection object.

```
void DisposeCollection(Collection target);
```

`target` A reference to the collection object you want to dispose of.

DESCRIPTION

The `DisposeCollection` function decrements the owner count of the collection object referenced by the `target` parameter. If the resulting owner count is 0, this function releases the memory occupied by the collection object, and the collection object reference contained in the `target` parameter becomes invalid.

The behavior of this function is undefined if you do not provide a reference to a valid collection object in the `target` parameter.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter "Introduction to QuickDraw GX" in *Inside Macintosh: QuickDraw GX Objects*.

For examples using this function, see "Creating or Disposing of a Collection" beginning on page 5-14.

To create a new collection object, use the `NewCollection` function, which is described on page 5-55.

To increment the owner count of a collection object, use the `CloneCollection` function, which is described in the next section. To determine the owner count of an existing collection object, use the `CountCollectionOwners` function, which is described on page 5-57.

Cloning and Copying Collection Objects

The functions described in this section allow you to examine and manipulate the owner count of a collection object or to make a complete copy of a collection object.

The `CloneCollection` function allows you to increment the owner count of a collection object. Typically, you use this function to signify the creation of a new reference to an existing collection object. The `CountCollectionOwners` function allows you to determine the current owner count of a collection object.

The `CopyCollection` function allows you to create a complete copy of a collection object. The new collection object contains a copy of every item in the original collection object.

CloneCollection

You can use the `CloneCollection` function to clone a collection object—that is, to increment its owner count.

```
Collection CloneCollection (Collection target);
```

`target` A reference to the collection object you want to clone.

function result A reference to the cloned collection. (This result is effectively a copy of the reference you provide in the `target` parameter.)

DESCRIPTION

The `CloneCollection` function increments the owner count of the collection object referenced by the `target` parameter, and, as a programming convenience, returns a reference to this collection as the function result.

Typically, you use this function to increment a collection object's owner count to represent a new reference to the collection object. For example, if you want two variables in your application to reference a single collection object, you can use this code to maintain the correct owner count:

```
firstReference = NewCollection();
secondReference = CloneCollection(firstReference);
```

Disposing of either reference (using the `DisposeCollection` function) simply decrements the collection's owner count. Disposing of the remaining reference decrements the owner count again and frees the memory associated with the collection.

The `CloneCollection` function does not return an error code.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter "Introduction to QuickDraw GX" in *Inside Macintosh: QuickDraw GX Objects*.

For examples of this function, see "Cloning or Copying a Collection" beginning on page 5-14.

To decrement the owner count of a collection object, use the `DisposeCollection` function, which is described in the previous section. To determine the owner count of an existing collection object, use the `CountCollectionOwners` function, which is described in the next section.

To copy a collection object, use the `CopyCollection` function, which is described on page 5-57.

CountCollectionOwners

You can use the `CountCollectionOwners` function to determine the number of existing references to a collection object.

```
long CountCollectionOwners(Collection source);
```

`source` The collection object whose owner count you want to determine.

function result The owner count of the collection object.

DESCRIPTION

The `CountCollectionOwners` function returns as its function result the owner count of the collection object referenced by the `source` parameter.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter "Introduction to QuickDraw GX Objects" in *Inside Macintosh: QuickDraw GX Objects*.

For examples of this function, see "Cloning or Copying a Collection" on page 5-14.

To increment the owner count of a collection object, use the `CloneCollection` function, which is described on page 5-56. To decrement the owner count of a collection object, use the `DisposeCollection` function, which is described on page 5-55.

CopyCollection

You use the `CopyCollection` function to create a copy of an existing collection.

```
Collection CopyCollection(Collection source, Collection target);
```

`source` A reference to the collection object you want to copy.

`target` A reference to a collection object to contain the copied collection items. You may provide `nil` for this parameter to request that the Collection Manager create a new collection object to hold the copied information.

function result A reference to the collection object containing the copied information.

DESCRIPTION

The `CopyCollection` function copies all of the information (except the owner count and exception procedure) from the collection object referenced by the `source` parameter into the collection object referenced by the `target` parameter.

If you specify `nil` for the `target` parameter, this function creates a new collection object to copy the information into. (This function does not return an error code; it returns `nil` if it cannot create a new collection object.)

In either case, this function returns a reference to the collection object containing the copied information.

SEE ALSO

For general information about QuickDraw GX objects, see the chapter "Introduction to QuickDraw GX Objects" in *Inside Macintosh: QuickDraw GX Objects*.

For examples using this function, see "Cloning or Copying a Collection" on page 5-14.

To clone a collection object, use the `CloneCollection` function, which is described on page 5-56.

Getting and Setting the Exception Procedure for a Collection

The functions described in this section allow you to examine and alter a collection object's exception procedure. You are allowed to specify an exception procedure for any collection object. When the Collection Manager encounters an error while operating on a collection object, it calls that collection's exception procedure, sending it the result code associated with the error.

The `GetCollectionExceptionProc` function allows you to obtain a pointer to the exception procedure intalled in a specified collection.

The `SetCollectionExceptionProc` function allows you to install a new exception procedure into a collection.

You can find a description of exception procedures on page 5-101.

GetCollectionExceptionProc

You use the `GetCollectionExceptionProc` function to obtain a pointer to the exception procedure installed in a specified collection.

```
CollectionExceptionProc GetCollectionExceptionProc
                        (Collection source);
```

`source` A reference to the collection object whose exception procedure you want to determine.

function result A pointer to the exception procedure installed in the source collection object.

DESCRIPTION

The `GetCollectionExceptionProc` function returns as its function result a pointer to the exception procedure installed in the collection object referenced by the `source` parameter.

SEE ALSO

To install a new exception procedure in a collection object, use the `SetCollectionExceptionProc` function, which is described in the next section.

For more information about exception procedures, see page 5-101.

SetCollectionExceptionProc

You use the `SetCollectionExceptionProc` function to install an exception procedure in a collection object.

```
void SetCollectionExceptionProc(Collection target,
      CollectionExceptionProc newExceptionProc);
```

`target` A reference to the collection object whose exception procedure you want to change.

`newExceptionProc`
A pointer to the new exception procedure.

DESCRIPTION

The `SetCollectionExceptionProc` function copies the function pointer from the `newExceptionProc` parameter into the collection object referenced by the `target` parameter.

SEE ALSO

For an example using this function, see “Installing an Exception Procedure” beginning on page 5-45.

To obtain a pointer to an existing exception procedure in a collection object, use the `GetCollectionExceptionProc` function, which is described in the previous section.

For more information about exception procedures, see page 5-101.

Getting and Setting the Default Attributes for a Collection

The functions described in this section allow you to examine and alter a collection object's default attributes. The default attributes of a collection specify the attributes that the Collection Manager assigns to new items added to the collection.

The `GetCollectionDefaultAttributes` function allows you to determine a collection's current default attributes. The `SetCollectionDefaultAttributes` function allows you to change a collection's default attributes.

GetCollectionDefaultAttributes

You use the `GetCollectionDefaultAttributes` function to examine the default attributes of a collection object.

```
long GetCollectionDefaultAttributes(Collection source);
```

`source` A reference to the collection object whose default attributes you want to determine.

function result A long word containing the bit flags that make up the collection's default attributes.

DESCRIPTION

The `GetCollectionDefaultAttributes` function returns as its function result the default attributes of the collection object referenced by the `source` parameter.

SEE ALSO

For information about default attributes for collection objects, see "Collection Attributes" beginning on page 5-9.

For information about attribute-related data types and enumerations, see page 5-49 through page 5-53.

To change the attributes of a collection object, use the `SetCollectionDefaultAttributes` function, which is described in the next section.

To examine the attributes of a specific item in a collection, use the functions described in "Getting Information About a Collection Item" beginning on page 5-76.

SetCollectionDefaultAttributes

You use the `SetCollectionDefaultAttributes` function to alter the default attributes of a collection object.

```
void SetCollectionDefaultAttributes(Collection target,
                                    long whichAttributes,
                                    long newAttributes);
```

`target` A reference to the collection object whose default attributes you want to alter.

`whichAttributes`
A mask indicating which bit flags in the target collection's default attributes you want to alter.

`newAttributes`
A long word containing the new values for the bit flags.

DESCRIPTION

The `SetCollectionDefaultAttributes` function copies the values of bit flags from the `newAttributes` parameter into the default attributes of the target collection.

This function uses the `whichAttributes` parameter to determine which bits to copy. For every bit in the `whichAttributes` parameter, this function takes one of two actions:

- If the bit is set, this function copies the value of the corresponding bit from the `newAttributes` parameter into the corresponding bit of the default attributes of the target collection.
- If the bit is not set, the corresponding bit of the target collection's default attributes remains unchanged.

SEE ALSO

For information about default attributes for collection objects, see "Collection Attributes" beginning on page 5-9.

For information about attribute-related data types and enumerations, see page 5-49 through page 5-53.

For examples of this function, see "Changing the Default Attributes of a Collection" beginning on page 5-15.

To examine the attributes of a collection object, use the `GetCollectionDefaultAttributes` function, which is described in the previous section.

To change the attributes of a specific item in a collection, use the functions described in "Editing Item Attributes" beginning on page 5-82.

Adding and Replacing Items in a Collection

The functions described in this section allow you to add items to a collection and replace items already in a collection.

The `AddCollectionItem` function allows you to add a new item to a collection. You can also use this function to replace a collection item by specifying its collection tag and collection ID.

The `ReplaceIndexedCollectionItem` function allows you to replace a collection item by specifying its collection index.

AddCollectionItem

You use the `AddCollectionItem` function to add a new item to a collection or to replace an existing item in a collection.

```
OSErr AddCollectionItem (Collection target,
                         CollectionTag tag, long id,
                         long itemSize, void *itemData);
```

`target`	A reference to the collection you want to add the item to.
`tag`	The collection tag you want to associate with the new item.
`id`	The collection ID you want to associate with the new item.
`itemSize`	The size in bytes of the item's variable-length data.
`itemData`	A pointer to the item's variable-length data.

DESCRIPTION

The `AddCollectionItem` function adds an item to the collection referenced by the `target` parameter. This new item contains

- the collection tag specified by the `tag` parameter
- the collection ID specified by the `id` parameter
- the attributes specified by the default attributes of the target collection
- the variable-length data specified by the `itemSize` and `itemData` parameters

This function copies the information pointed to by the `itemData` parameter into the new item; after calling this function, you may alter this information or free the memory pointed to by this parameter without affecting the collection.

If the target collection already contains an item with the same collection tag and collection ID as specified in the `tag` and `id` parameters, this function removes the original item and replaces it with the new one, unless the existing item is locked. If it is locked, this function returns a `collectionItemLockedErr` result code.

The `itemSize` parameter determines how many bytes of information this function copies into the new item. If you specify 0 for this parameter, or provide `nil` for the `itemData` parameter, this function copies no information into the variable-length data of the new item, or removes the variable-length data if the item already exists.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemLockedErr`	–5750	Can't replace locked item.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

For information about locking collection items, see "Getting and Setting the Attributes of an Item" beginning on page 5-24. To lock a collection item, use the functions described in "Editing Item Attributes" beginning on page 5-82.

For examples using this function, see "Adding Items to a Collection" beginning on page 5-17 and "Replacing Items in a Collection" beginning on page 5-28.

To replace a collection item using the item's index (rather than the item's tag and ID), use the `ReplaceIndexedCollectionItem` function, described in the next section.

To remove an item from a collection, use the functions described in "Removing Items From a Collection" beginning on page 5-65.

ReplaceIndexedCollectionItem

You use the `ReplaceIndexedCollectionItem` function to replace the variable-length data of an item in a collection given the item's index.

```
OSErr ReplaceIndexedCollectionItem(Collection target, long index,
                                   long itemSize, void *itemData);
```

`target`	A reference to the collection containing the item you want to replace.
`index`	The collection index associated with the item to replace.
`itemSize`	The item's size.
`itemData`	A pointer to the item's data.

DESCRIPTION

The `ReplaceIndexedCollectionItem` function replaces the variable-length data associated with an item in the target collection. You specify which item to replace using the `index` parameter. If the target collection does not contain an item whose collection index matches the value of the `index` parameter, this function returns a `collectionIndexRangeErr` result code.

If the target collection does contain an item with the specified index, this function replaces that item with a new item (if the existing item is not locked—if it is, this function returns a `collectionItemLockedErr` result code). The new item contains

- the same collection tag as the original item
- the same collection ID as the original item
- the same attributes as the original item
- the variable-length data specified by the `itemSize` and `itemData` parameters

This function copies the information pointed to by the `itemData` parameter into the new item; after calling this function, you may alter this information or free the memory pointed to by this parameter without affecting the collection.

The `itemSize` parameter determines how many bytes of information this function copies into the new item. If you specify 0 for this parameter, or provide `nil` for the `itemData` parameter, this function copies no information into the variable-length data of the new item, or removes the variable-length data if the item already exists.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemLockedErr`	–5750	Can't replace locked item.
`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

For information about locking collection items, see "Getting and Setting the Attributes of an Item" beginning on page 5-24. To lock a collection item, use the functions described in "Editing Item Attributes" beginning on page 5-82.

To replace a collection item using the item's tag and ID (rather than the item's index), use the `ReplaceIndexedCollectionItem` function, described on page 5-63.

To remove an item from a collection, use the functions described in the next section.

Removing Items From a Collection

The functions described in this section allow you to remove items from a collection.

The `RemoveCollectionItem` and `RemoveIndexedCollectionItem` functions allow you to remove a single item from a collection. You use the `RemoveCollectionItem` function if you want to specify the item to remove using the item's tag and ID. You use the `RemoveIndexedCollectionItem` function if you want to specify the item to remove using the item's index.

The `PurgeCollection` function allows you to remove from a collection all the items whose attributes match a specified pattern.

The `PurgeCollectionTag` function allows you to remove from a collection all the items with a specified collection tag.

The `EmptyCollection` function allows you to remove every item from a collection.

RemoveCollectionItem

You can use the `RemoveCollectionItem` function to remove an item from a collection given the item's associated collection tag and collection ID.

```
OSErr RemoveCollectionItem (Collection target,
                            CollectionTag tag, long id);
```

`target`	A reference to the collection object from which you want to remove the item.
`tag`	The collection tag associated with the item you want to remove.
`id`	The collection ID associated with the item you want to remove.

DESCRIPTION

The `RemoveCollectionItem` function removes the item specified by the `tag` and `id` parameters from the collection referenced by the `target` parameter. This function removes the specified item even if its lock attribute is set.

If the target collection does not contain an item whose collection tag and collection ID match the values in the `tag` and `id` parameters, this function returns a `collectionItemNotFoundErr` result code.

RESULT CODES

`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples using this function, see “Removing Items From a Collection” beginning on page 5-30.

To remove a collection item using the item’s index (rather than the item’s tag and ID), use the `RemoveIndexedCollectionItem` function, described in the next section.

To replace an item in a collection, use the functions described in “Adding and Replacing Items in a Collection” beginning on page 5-62.

RemoveIndexedCollectionItem

You can use the `RemoveIndexedCollectionItem` function to remove an item from a collection given the item’s index.

```
OSErr RemoveIndexedCollectionItem(Collection target, long index);
```

`target` A reference to the collection object from which you want to remove the item.

`index` The collection index of the item you want to remove.

DESCRIPTION

The `RemoveIndexedCollectionItem` function removes the item specified by the `index` parameter from the collection referenced by the `target` parameter. This function removes the specified item even if its lock attribute is set.

If the target collection does not contain an item whose collection index matches the values in the `index` parameter, this function returns a `collectionIndexRangeErr` result code.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples using this function, see “Removing Items From a Collection” beginning on page 5-30.

To remove a collection item using the item’s tag and ID (rather than the item’s index), use the `RemoveCollectionItem` function, described in the previous section.

To replace an item in a collection, use the functions described in “Adding and Replacing Items in a Collection” beginning on page 5-62.

PurgeCollection

You use the `PurgeCollection` function to remove all items in a collection whose attributes match a specified pattern.

```
void PurgeCollection(Collection target,
                     long whichAttributes,
                     long matchingAttributes);
```

`target` A reference to the collection object containing the items you want to remove.

`whichAttributes`
A mask indicating which attributes you want to test.

`matchingAttributes`
A long word containing the values of the attributes you want to match.

DESCRIPTION

The `PurgeCollection` function removes from the target collection any items whose attributes match the criteria you specify in the `whichAttributes` and `matchingAttributes` parameters.

The `whichAttributes` parameter allows you to specify which attributes this function examines. You should set the bits of the `whichAttributes` parameter that correspond to the attributes you want to test.

This function compares the specified attributes of each item in the target collection with the corresponding attributes in the `matchingAttributes` parameter. If the values of all the specified attributes match, the function removes the item. To avoid purging locked items, you should clear the lock attribute in the `whichAttributes` and `matchingAttributes` parameters.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples using this function, see “Removing Items From a Collection” beginning on page 5-30.

To remove all of the items in a collection with a specified collection tag, use the `PurgeCollectionTag` function, described in the next section.

To remove every item in a collection, use the `EmptyCollection` function, described on page 5-68.

PurgeCollectionTag

You use the `PurgeCollectionTag` function to remove from a collection all items with a specific collection tag.

```
void PurgeCollectionTag(Collection target,
                        CollectionTag tag);
```

`target` A reference to the collection object containing the items you want to remove.

`tag` The collection tag associated with the items to remove.

DESCRIPTION

The `PurgeCollectionTag` function removes from the target collection all items whose collection tag matches the value of the `tag` parameter. This function removes locked and unlocked items.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples using this function, see “Removing Items From a Collection” beginning on page 5-30.

To remove all of the items in a collection whose attributes match a specified pattern, use the `PurgeCollection` function, described in the previous section.

To remove every item in a collection, use the `EmptyCollection` function, described in the next section.

EmptyCollection

You use the `EmptyCollection` function to remove every item in a collection.

```
void EmptyCollection (Collection target);
```

`target` A reference to the collection object you want to empty.

DESCRIPTION

This function removes every item in the collection referenced by the `target` parameter. This function provides the fastest mechanism for emptying a collection.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

To remove all of the items in a collection whose attributes match a specified pattern, use the `PurgeCollection` function, described on page 5-67.

To remove all of the items in a collection with a specified collection tag, use the `PurgeCollectionTag` function, described in the previous section.

Counting Items in a Collection

The functions described in this section allow you to count items in a collection.

The `CountCollectionItems` function allows you to determine the total number of items in a collection.

The `CountTaggedCollectionItems` function allows you to determine the total number of items in a collection that have a specified collection tag.

CountCollectionItems

You can use the `CountCollectionItems` function to determine the total number of items in a collection.

```
long CountCollectionItems(Collection source);
```

`source` A reference to the collection object whose items you want to count.

function result The total number of items in the source collection.

DESCRIPTION

The `CountCollectionItems` function returns as its function result the total number of items in the collection referenced by the `source` parameter.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

For examples using this function, see "Adding Items to a Collection" beginning on page 5-17.

To count the items in a collection that have a specified collection tag, use the `CountTaggedCollectionItems` function, described in the next section.

CountTaggedCollectionItems

You can use the `CountTaggedCollectionItems` function to obtain the total number of items in a collection that have a specified collection tag.

```
long CountTaggedCollectionItems(Collection source,
                                CollectionTag tag);
```

`source` A reference to the collection object whose items you want to count.

`tag` The collection tag associated with the items you want to count.

function result The total number of items in the source collection whose collection tags match the value specified in the `tag` parameter.

DESCRIPTION

The `CountTaggedCollectionItems` function returns as its function result the total number of items in the source collection whose collection tags match the value you specify in the `tag` parameter.

SEE ALSO

For information about collection items, see “Collection Items” beginning on page 5-8.

For examples of this function, see “Adding Items to a Collection” beginning on page 5-17.

To count all of the items in a collection, use the `CountCollectionItems` function, described in the previous section.

Retrieving the Variable-Length Data From an Item

The functions described in this section allow you to obtain a copy of the variable-length data associated with a specified collection item.

The `GetCollectionItem` function allows you to retrieve data from an item given its collection tag and collection ID. The `GetIndexedCollectionItem` function allows you to retrieve data from an item given its collection index.

The `GetTaggedCollectionItem` function provides another way for you to specify the item whose data you want to retrieve. With this function, you specify the item using the item’s collection tag and the item’s tag list position. See “Methods of Identifying Collection Items” beginning on page 5-11 for a discussion of collection tags and tag list positions.

GetCollectionItem

You can use the `GetCollectionItem` function to obtain a copy of the variable-length data associated with a collection item given the item's collection tag and collection ID.

```
OSErr GetCollectionItem(Collection source,
                        CollectionTag tag,
                        long id,
                        long *itemSize,
                        void *itemData);
```

`source` A reference to the collection object containing the item whose data you want to retrieve.

`tag` The collection tag associated with the item whose data you want to retrieve.

`id` The collection ID associated with the item whose data you want to retrieve.

`itemSize` A pointer to a `long` value indicating the number of bytes of data you want returned in the `itemData` parameter. On return, this value indicates the size in bytes of the variable-length data associated with the specified item. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all the specified item's variable-length data and you do not want to determine the size of this data.

`itemData` A pointer to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item. You may specify the constant `dontWantData` for this parameter if you do not want a copy of the item's data.

DESCRIPTION

The `GetCollectionItem` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter and you specify an item in that collection using the `tag` and `id` parameters.

You use the `itemSize` parameter to specify how many bytes of data to return in the `itemData` parameter. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all of the variable-length data from the specified item into the `itemData` parameter. You may specify a value for the `itemSize` parameter that is greater than the actual number of bytes in the specified item's variable-length data; however, this function never returns in the `itemData` parameter more data than contained in the specified item's variable-length data.

This function returns information in the `itemSize` and `itemData` parameters:

- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

If you don't know the size of the item you want to retrieve, you typically call this function twice. The first time you provide a pointer in the `itemSize` parameter to determine the size of the specified item's data and you specify `dontWantData` for the `itemData` parameter. Then you allocate a memory block large enough to hold a copy of the item's data. Then you call the function a second time. This time you specify the constant `dontWantSize` for the `itemSize` parameter and provide a pointer to the allocated memory block for the `itemData` parameter. The function then copies the data into the allocated block of memory.

RESULT CODES

`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8.

For examples using this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item given its collection index (rather than its collection tag and ID), use the `GetIndexedCollectionItem` function, described in the next section.

GetIndexedCollectionItem

You can use the `GetIndexedCollectionItem` function to obtain a copy of the variable-length data associated with a collection item given the item's collection index.

```
OSErr GetIndexedCollectionItem(Collection source,
                               long index,
                               long *itemSize,
                               void *itemData);
```

`source`	A reference to the collection object containing the item whose data you want to retrieve.
`index`	The collection index associated with the item whose data you want to retrieve.
`itemSize`	A pointer to a `long` value indicating the number of bytes of data you want returned in the `itemData` parameter. On return, this value indicates the size in bytes of the variable-length data associated with the specified item. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all of the specified item's variable-length data and you do not want to determine the size of this data.
`itemData`	A pointer to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item. You may specify the constant `dontWantData` for this parameter if you do not want a copy of the item's data.

DESCRIPTION

The `GetIndexedCollectionItem` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter and you specify an item in that collection using the `index` parameter.

You use the `itemSize` parameter to specify how many bytes of data to return in the `itemData` parameter. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all of the variable-length data from the specified item into the `itemData` parameter. You may specify a value for the `itemSize` parameter that is greater than the actual number of bytes in the specified item's variable-length data; however, this function never returns in the `itemData` parameter more data than contained in the specified item's variable-length data.

This function returns information in the `itemSize` and `itemData` parameters:

- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

If you don't know the size of the item you want to retrieve, you typically call this function twice. The first time you provide a pointer in the `itemSize` parameter to determine the size of the specified item's data and you specify the constant `dontWantData` for the `itemData` parameter. Then you allocate a memory block large enough to hold a copy of the item's data. Then you call the function a second time. This time you specify the constant `dontWantSize` for the `itemSize` parameter and provide a pointer to the allocated memory block for the `itemData` parameter. The function then copies the data into the allocated block of memory.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items and their associated variable-length data, see "Collection Items" beginning on page 5-8. For information about collection indexes, see "Methods of Identifying Collection Items" beginning on page 5-11.

For examples using this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item given its collection tag and ID (rather than its collection index), use the `GetCollectionItem` function, described in the previous section.

GetTaggedCollectionItem

You can use the `GetTaggedCollectionItem` function to obtain a copy of the variable-length data associated with a collection item given the item's collection tag and tag list position.

```
OSErr GetTaggedCollectionItem(Collection source,
                              CollectionTag tag,
                              long position,
                              long *itemSize,
                              void *itemData);
```

`source` A reference to the collection object containing the item whose data you want to retrieve.

`tag` The collection tag associated with the item whose data you want to retrieve.

`position` The tag list position associated with the specific item.

`itemSize` A pointer to a `long` value indicating the number of bytes of data you want returned in the `itemData` parameter. On return, this value indicates the size in bytes of the variable-length data associated with the specified item. You may specify the constant `dontWantSize` for this parameter to indicate that you want to copy all of the specified item's variable-length data and you do not want to determine the size of this data.

`itemData` A pointer to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item. You may specify the constant `dontWantData` for this parameter if you do not want a copy of the item's data.

DESCRIPTION

The `GetTaggedCollectionItem` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter; you specify the item in that collection using the `tag` and `position` parameters. In the `tag` parameter you specify the collection tag of the desired item and in the `position` parameter you specify the tag list position of the desired item.

Remember that a tag list position is the sequential index that determines an item given a specific collection tag. For example:

- A tag list position of 1 indicates the first item with the specified tag.
- A tag list position of 2 indicates the second item with the specified tag.

By sequentially incrementing the `position` parameter, you can use this function to step through all of the items in a collection without knowing their collection IDs.

This function returns information in the `itemSize` and `itemData` parameters:

- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

If you don't know the size of the item you want to retrieve, you typically call this function twice. The first time you provide a pointer in the `itemSize` parameter to determine the size of the specified item's data and you specify the constant `dontWantData` for the `itemData` parameter. Then you allocate a memory block large enough to hold a copy of the item's data. Then you call the function a second time. This time you specify the constant `dontWantSize` for the `itemSize` parameter and provide a pointer to the allocated memory block for the `itemData` parameter. The function then copies the data into the allocated block of memory.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items and their associated collection tags and variable-length data, see "Collection Items" beginning on page 5-8. For information about tag list positions, see "Methods of Identifying Collection Items" beginning on page 5-11.

For examples of this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item given its collection tag and ID, use the `GetCollectionItem` function, described on page 5-71.

To retrieve the data associated with a collection item given its collection index, use the `GetIndexedCollectionItem` function, described in the previous section.

Getting Information About a Collection Item

The functions described in this section allow you to determine information about a collection item, such as the item's collection index, the item's size, and the item's attributes.

Each function in this section provides a different way for you to specify which collection item you want to examine:

- The `GetCollectionItemInfo` function requires you to specify the item's collection tag and collection ID.
- The `GetIndexedCollectionItemInfo` function requires you to specify the item's collection index.
- The `GetTaggedCollectionItemInfo` function requires you to specify the item's collection tag and tag list position.

GetCollectionItemInfo

You use the `GetCollectionItemInfo` function to obtain information about a specific collection item given the item's collection tag and collection ID.

```
OSErr GetCollectionItemInfo(Collection source,
                            CollectionTag tag,
                            long id,
                            long *index,
                            long *itemSize,
                            long *attributes);
```

`source` A reference to the collection object containing the item you want to obtain information about.

`tag` The collection tag associated with the item you want to obtain information about.

`id` The collection ID associated with the item you want to obtain information about.

`index` A pointer to a `long` value. On return, this value represents the collection index of the specified item. You may specify the constant `dontWantIndex` for this parameter if you do not want to determine the specified item's collection index.

`itemSize` A pointer to a `long` value. On return, this value indicates the size in bytes of the variable-length data associated with the specified item. You may specify the constant `dontWantSize` for this parameter to indicate that you do not want to determine the size of this data.

`attributes`
A pointer to a `long` value. On return, this value contains a copy of the attributes associated with the specified item. You may specify the constant `dontWantAttributes` for this parameter if you do not want a copy of the item's attributes.

DESCRIPTION

The `GetCollectionItemInfo` function allows you to obtain information about a specific collection item in the collection referenced by the `source` parameter. You specify the collection item by specifying the item's collection tag and collection ID in the `tag` and `id` parameters.

This function returns information in the `index`, `itemSize`, and `attributes` parameters:

- If you provide a pointer in the `index` parameter, the function uses this parameter to return the collection index of the specified item. Once you have determined an item's collection index, you can use it to specify the item when calling Collection Manager functions, rather than using the item's collection tag and collection ID. Specifying collection items using their collection index, rather than using the item's collection tag and collection ID, generally results in improved performance.
- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `attributes` parameter, the function uses this parameter to return a copy of the attributes associated with the specified collection item.

RESULT CODES

`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8.

For examples of this function, see "Determining the Collection Index of an Item" beginning on page 5-19, "Determining the Size of an Item's Variable-Length Data" beginning on page 5-22, and "Getting and Setting the Attributes of an Item" beginning on page 5-24.

To obtain information about a collection item using the collection index to specify the item, use the `GetIndexedCollectionItemInfo` function, described in the next section.

To obtain information about a collection item using the collection tag and tag list position to specify the item, use the `GetTaggedCollectionItemInfo` function, described on page 5-80.

GetIndexedCollectionItemInfo

You use the `GetIndexedCollectionItemInfo` function to obtain information about a specific collection item given the item's collection index.

```
OSErr GetIndexedCollectionItemInfo (Collection source,
                                    long index,
                                    CollectionTag *tag,
                                    long *id,
                                    long *itemSize,
                                    long *attributes);
```

`source` A reference to the collection object containing the item you want to obtain information about.

`index` The collection index associated with the item you want to obtain information about.

`tag` A pointer to a collection tag. On return, the collection tag associated with the specified item. You may specify the constant `dontWantTag` for this parameter if you do not want to determine the specified item's collection tag.

`id` A pointer to a `long` value. On return, the collection ID associated with the specified item. You may specify the constant `dontWantId` for this parameter if you do not want to determine the specified item's collection ID.

`itemSize` A pointer to a `long` value. On return, this value indicates the size in bytes of the data associated with the specified item. You may specify the constant `dontWantSize` for this parameter if you do not want to determine the specified item's data size.

`attributes`
A pointer to a `long` value. On return, this value contains a copy of the attributes associated with the specified item. You may specify the constant `dontWantAttributes` for this parameter if you do not want a copy of the item's attributes.

DESCRIPTION

The `GetIndexedCollectionItemInfo` function allows you to obtain information about a specific collection item in the collection referenced by the `source` parameter. You specify the collection item by specifying the item's collection index in the `index` parameter.

This function returns information in the `tag`, `id`, `itemSize`, and `attributes` parameters:

- If you provide a pointer in the `tag` parameter, the function uses this parameter to return the collection tag of the specified item.
- If you provide a pointer in the `id` parameter, the function uses this parameter to return the collection ID of the specified item.
- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `attributes` parameter, the function uses this parameter to return a copy of the attributes associated with the specified collection item.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8. For information about collection indexes, see "Methods of Identifying Collection Items" beginning on page 5-11.

For examples of this function, see "Determining the Collection Index of an Item" beginning on page 5-19, "Determining the Size of an Item's Variable-Length Data" beginning on page 5-22, and "Getting and Setting the Attributes of an Item" beginning on page 5-24.

To obtain information about a collection item using the collection tag and collection ID to specify the item, use the `GetCollectionItemInfo` function, described in the previous section.

To obtain information about a collection item using the collection tag and tag list position to specify the item, use the `GetTaggedCollectionItemInfo` function, described in the next section.

GetTaggedCollectionItemInfo

You use the `GetTaggedCollectionItemInfo` function to obtain information about a specific collection item given the item's collection tag and tag list position.

```
OSErr GetTaggedCollectionItemInfo(Collection source,
                                  CollectionTag tag,
                                  long position,
                                  long *id,
                                  long *index,
                                  long *itemSize,
                                  void *attributes);
```

`source` A reference to the collection object containing the item you want to obtain information about.

`tag` The collection tag associated with the item you want to obtain information about.

`position` The tag list position of the item you want to obtain information about.

`id` A pointer to a `long` value. On return, this value represents the collection ID associated with the specified item. You may specify the constant `dontWantId` for this parameter if you do not want to determine the specified item's collection ID.

`index` A pointer to a `long` value. On return, this value represents the collection index of the specified item. You may specify the constant `dontWantIndex` for this parameter if you do not want to determine the specified item's collection index.

`itemSize` A pointer to a `long` value. On return, this value indicates the size in bytes of the data associated with the specified item. You may specify the constant `dontWantSize` for this parameter if you do not want to determine the specified item's data size.

`attributes`
A pointer to a `long` value. On return, this value contains a copy of the attributes associated with the specified item. You may specify the constant `dontWantAttributes` for this parameter if you do not want a copy of the item's attributes.

DESCRIPTION

The `GetTaggedCollectionItemInfo` function allows you to obtain information about a specific collection item in the collection referenced by the source parameter. You specify the item in the source collection using the `tag` and `position` parameters. In the `tag` parameter you specify the collection tag of the desired item and in the `position` parameter you specify the tag list position of the desired item.

Remember that a collection tag and a tag list position uniquely identify a collection item. The tag list position indicates where the collection item would lie in a list made up of all the collection items with the same collection tag. For example:

- A tag list position of 1 indicates the first item with the specified tag.
- A tag list position of 2 indicates the second item with the specified tag.

By sequentially incrementing the `position` parameter, you can use this function to step through all of the items in a collection that share a collection tag without knowing their collection IDs.

The `GetTaggedCollectionItemInfo` function returns information in the `id`, `index`, `itemSize`, and `attributes` parameters:

- If you provide a pointer in the `id` parameter, the function uses this parameter to return the collection ID of the specified item.
- If you provide a pointer in the `index` parameter, the function uses this parameter to return the collection index of the specified item.
- If you provide a pointer in the `itemSize` parameter, the function uses this parameter to return the size in bytes of the variable-length data associated with the specified collection item.
- If you provide a pointer in the `attributes` parameter, the function uses this parameter to return a copy of the attributes associated with the specified collection item.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see “Collection Items” beginning on page 5-8. For information about tag list positions, see “Methods of Identifying Collection Items” beginning on page 5-11.

For examples of this function, see “Determining the Collection Index of an Item” beginning on page 5-19, “Determining the Size of an Item’s Variable-Length Data” beginning on page 5-22, and “Getting and Setting the Attributes of an Item” beginning on page 5-24.

To obtain information about a collection item using the collection tag and collection ID to specify the item, use the `GetCollectionItemInfo` function, described on page 5-76.

To obtain information about a collection item using the collection index to specify the item, use the `GetIndexedCollectionItemInfo` function, described in the previous section.

Editing Item Attributes

The functions described in this section allow you to edit the attributes of a collection item. Each function in this section provides a different way for you to specify the collection item whose attributes you want to edit:

- The `SetCollectionItemInfo` function requires you to specify the item's collection tag and collection ID.
- The `SetIndexedCollectionItemInfo` function requires you to specify the item's collection index.

SetCollectionItemInfo

You use the `SetCollectionItemInfo` function to edit the attributes of a specific collection item given the item's collection tag and collection ID.

```
OSErr SetCollectionItemInfo(Collection target,
                            CollectionTag tag,
                            long id,
                            long whichAttributes,
                            long newAttributes);
```

`target` A reference to the collection object containing the item whose attributes you want to edit.

`tag` The collection tag associated with the item whose attributes you want to edit.

`id` The collection ID associated with the item whose attributes you want to edit.

`whichAttributes`
A mask indicating which attributes you want to edit.

`newAttributes`
A long word containing the new settings for the attributes.

DESCRIPTION

The `SetCollectionItemInfo` function allows you to edit the attributes of a specific collection item in the collection referenced by the `target` parameter. You specify the collection item by specifying the item's collection tag and collection ID in the `tag` and `id` parameters.

This function copies bit values from the `newAttributes` parameter to the attributes associated with the specified item.

This function uses the `whichAttributes` parameter to determine which bits to copy. For every bit in the `whichAttributes` parameter, this function takes one of two actions:

- If the bit is set, this function copies the value of the corresponding bit from the `newAttributes` parameter into the corresponding bit of the attributes associated with the specified item.
- If the bit is not set, the corresponding bit of the specified item's attributes remains unchanged.

The `whichAttributes` parameter allows you to change the values of specific bits in the specified item's attributes without affecting the values of other bits.

RESULT CODES

`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection attributes, see "Collection Attributes" beginning on page 5-9.

For attribute-related data types and enumerations, see page 5-49 through page 5-53.

For examples of this function, see "Getting and Setting the Attributes of an Item" beginning on page 5-24.

To obtain information about a collection item using the collection index to specify the item, use the `SetIndexedCollectionItemInfo` function, described in the next section.

SetIndexedCollectionItemInfo

You use the `SetIndexedCollectionItemInfo` function to edit the attributes of a specific collection item given the item's collection index.

```
OSErr SetIndexedCollectionItemInfo(Collection target,
                                   long index,
                                   long whichAttributes,
                                   long newAttributes);
```

`target` A reference to the collection object containing the item whose attributes you want to edit.

`index` The collection index of the item whose attributes you want to edit.

`whichAttributes`
A mask indicating which attributes you want to edit.

`newAttributes`
A long word containing the new settings for the attributes.

DESCRIPTION

The `SetIndexedCollectionItemInfo` function allows you to edit the attributes of a specific collection item in the collection referenced by the `target` parameter. You specify the collection item by specifying the item's collection index in the `index` parameter.

This function copies bit values from the `newAttributes` parameter to the attributes associated with the specified item.

This function uses the `whichAttributes` parameter to determine which bits to copy. For every bit in the `whichAttributes` parameter, this function takes one of two actions:

- If the bit is set, this function copies the value of the corresponding bit from the `newAttributes` parameter into the corresponding bit of the attributes associated with the specified item.
- If the bit is not set, the corresponding bit of the specified item's attributes remains unchanged.

The `whichAttributes` parameter allows you to change the values of specific bits in the specified item's attributes without affecting the values of other bits.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection attributes, see “Collection Attributes” beginning on page 5-9.

For attribute-related data types and enumerations, see page 5-49 through page 5-53.

For examples of this function, see “Getting and Setting the Attributes of an Item” beginning on page 5-24.

To edit the attributes of collection item using the collection tag and collection ID (rather than the collection index) to specify the item, use the `SetCollectionItemInfo` function, described in the previous section.

To examine the attributes of a collection item, use the functions described in “Getting Information About a Collection Item” beginning on page 5-76.

Getting Information About Collection Tags

You use the `CollectionTagExists` function to identify if a specific collection tag exists within a collection. You use the `CountCollectionTags` function to obtain the number of unique collection tags in a collection.

You use the `GetIndexedCollectionTag` function to obtain a specific collection tag from a collection.

CollectionTagExists

You can use the `CollectionTagExists` function to identify if any of the items in a specified collection contain a specified collection tag.

```
Boolean CollectionTagExists(Collection source,
                            CollectionTag tag);
```

`source` A reference to the collection object you want to search for a specific collection tag.

`tag` The collection tag to search for in the collection.

function result A Boolean value indicating whether the source collection contains any items that contain the specified tag.

DESCRIPTION

The `CollectionTagExists` function returns as its function result a Boolean value indicating whether any of the items in the collection referenced by the `source` parameter contain the collection tag specified by the `tag` parameter.

SEE ALSO

For information about collection tags, see “Collection Items” beginning on page 5-8. For information about data types related to collection tags, see the section “Collection Tags” on page 5-49.

CountCollectionTags

You use the `CountCollectionTags` function to determine the number of distinct collection tags contained by the items of a specified collection.

```
long CountCollectionTags(Collection source);
```

`source` A reference to the collection object whose collection tags you want to count.

function result The number of distinct collection tags contained by the items of the source collection.

DESCRIPTION

The `CountCollectionTags` function returns as its function result the number of distinct collection tags contained by the items of the collection referenced by the `source` parameter.

SEE ALSO

For information about collection tags, see “Collection Items” beginning on page 5-8. For information about data types related to collection tags, see the section “Collection Tags” on page 5-49.

For an example of this function, see “Examining the Collection Tags of a Collection” beginning on page 5-35.

GetIndexedCollectionTag

Each collection object contains a number of distinct collection tags. You can use the `GetIndexedCollectionTag` function to examine a specific collection tag contained in a collection.

```
OSErr GetIndexedCollectionTag(Collection source,
                              long whichTag,
                              CollectionTag *tag);
```

`source`	The collection from which to obtain a specific collection tag.
`whichTag`	The position of the desired collection tag in the source collection's list of distinct collection tags.
`tag`	A pointer to a collection tag. On return, the collection tag that lies at the specified position in the list of distinct collection tags contained in the source collection.

DESCRIPTION

The `GetIndexedCollectionTag` function returns in the `tag` parameter the collection tag that lies at the position specified by the `whichTag` parameter in the list of distinct collection tags contained in the collection referenced by the `source` parameter.

By sequentially incrementing the value of the `whichTag` parameter from 1 to the result of the `CountCollectionTags` function, you can use this function to determine every collection tag contained in a collection.

RESULT CODES

`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection tags, see "Collection Items" beginning on page 5-8. For information about data types related to collection tags, see the section "Collection Tags" beginning on page 5-49.

For an example of this function, see "Examining the Collection Tags of a Collection" beginning on page 5-35.

To determine the total number of distinct collection tags contained in a collection, use the `CountCollectionTags` function, described in the previous section.

Flattening and Unflattening a Collection

You use the `FlattenCollection` function to flatten a collection into a stream of bytes. You use the `UnflattenCollection` function to unflatten a collection that was flattened using the `FlattenCollection` function.

FlattenCollection

You can use the `FlattenCollection` function to convert a collection object into a stream format suitable for storing and unflattening. For example, you could use this function to copy a collection onto the Clipboard so that it could be pasted into another application.

```
OSErr FlattenCollection(Collection source,
                        CollectionFlattenProc flattenProc,
                        void *refCon);
```

`source` A reference to the collection that you want to flatten.

`flattenProc`
A pointer to a callback function you provide to process the flattened stream of bytes.

`refCon` A reference constant that you want the Collection Manager to pass repeatedly to the callback function.

DESCRIPTION

The `FlattenCollection` function flattens into a stream of bytes the collection you specify with the `source` parameter. As this function flattens the collection, it repeatedly calls the callback function you specify using the `flattenProc` parameter. Each time it calls this function, it provides the callback function with a pointer to a block of memory containing flattened data. It continues to call this function until it has flattened the entire collection. Your callback function can process the flattened data in a number of ways: it could copy the flattened data into a handle-based block of memory, it could write the flattened data to disk, and so on.

In the `refCon` parameter, you specify a value that the Collection Manager passes on to your callback function each time it calls your callback function. You can use this parameter as a pointer to a structure containing information your callback function needs to process the blocks of flattened data.

When flattening the source collection, this function includes only the collection items whose persistence attribute is set.

This function can return any error returned by the callback function.

SEE ALSO

For information about the persistence attribute, see "Collection Items" beginning on page 5-8.

For information about the callback function that you provide, see page 5-100.

For examples of this function, see "Flattening and Unflattening a Collection" beginning on page 5-37 and "Reading Collections From and Writing Collections to Disk" beginning on page 5-41.

To create a flattened collection that includes only those collection items whose attributes match a specified pattern, use the `FlattenPartialCollection` function, described in the next section.

To unflatten a flattened collection, use the `UnflattenCollection` function, described on page 5-90.

FlattenPartialCollection

You can use the `FlattenPartialCollection` function to convert a collection object into a stream format suitable for storage and unflattening. With this function, you can include in the flattened collection only those items whose attributes match a specified pattern.

```
OSErr FlattenPartialCollection(Collection source,
                               CollectionFlattenProc flattenProc,
                               void *refCon,
                               long whichAttributes,
                               long matchingAttributes)
```

`source` The collection that you want to flatten.

`flattenProc`
A pointer to a function to write data.

`refCon` A reference constant that you want the Collection Manager to pass repeatedly to the flatten procedure.

`whichAttributes`
A mask indicating which attributes you want to test.

`matchingAttributes`
A long word containing the attribute values you want to match.

DESCRIPTION

The `FlattenPartialCollection` function flattens into a stream of bytes the collection you specify with the `source` parameter. It includes only the collection items whose attributes specified by the `whichAttributes` parameter match the values specified by the `matchingAttributes` parameter.

As this function flattens the collection, it repeatedly calls the callback function you specify using the `flattenProc` parameter. Each time it calls this function, it provides the callback function with a pointer to a block of memory containing flattened data. It continues to call this function until it has flattened the entire collection. Your callback function can process the flattened data in a number of ways: it could copy the flattened data into a handle-based block of memory, it could write the flattened data to disk, and so on.

In the `refCon` parameter, you specify a value that the Collection Manager passes on to your callback function each time it calls your callback function. You can use this parameter as a pointer to a structure containing information your callback function needs to process the blocks of flattened data.

When flattening the source collection, this function includes only the collection items whose persistence attribute is set, regardless of the values you provide in the `whichAttributes` and `matchingAttributes` parameters.

This function can return any error returned by the callback function.

SEE ALSO

For information about matching collection item attributes, see “Collection Items” beginning on page 5-8.

For information about the callback function that you provide, see page 5-100.

To create a flattened collection that includes every item in a collection, use the `FlattenCollection` function, described in the previous section.

To unflatten a flattened collection, use the `UnflattenCollection` function, described in the next section.

UnflattenCollection

You use the `UnflattenCollection` function to unflatten a collection that was flattened using the `FlattenCollection` or `FlattenPartialCollection` function.

```
OSErr UnflattenCollection (Collection target,
                           CollectionFlattenProc flattenProc,
                           void *refCon);
```

`target` A reference to the collection object you want to create from the flattened data.

`flattenProc` A pointer to a function to read in flattened data.

`refCon` A reference constant that you want the Collection Manager to pass repeatedly to the callback function.

DESCRIPTION

The `UnflattenCollection` function unflattens a stream of bytes into the collection object you specify with the `target` parameter.

As this function unflattens the collection, it repeatedly calls the callback function you specify using the `flattenProc` parameter. Each time it calls this function, it provides the callback function with a pointer to a block of memory and a requested size. The callback function is responsible for reading the next set of bytes from the flattened byte stream and copying the data into the block of memory.

The Collection Manager continues to call your callback function, requesting more of the flattened stream of bytes each time, until it has unflattened the entire collection. Your callback function can read the flattened data from any source you choose: it could read the flattened data from a handle-based block of memory, it could read the flattened data from disk, and so on.

In the `refCon` parameter, you specify a value that the Collection Manager passes on to your callback function each time it calls your callback function. You can use this parameter as a pointer to a structure containing information your callback function needs when reading the blocks of flattened data.

This function can return any error returned by the callback function.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionVersionErr`	–5753	Unrecognized version / data may be corrupt.

SEE ALSO

For examples of this function, see "Flattening and Unflattening a Collection" beginning on page 5-37 and "Reading Collections From and Writing Collections to Disk" beginning on page 5-41.

For information about the callback function that you provide, see page 5-100.

To create a flattened collection that includes only those collection items whose attributes match a specified pattern, use the `FlattenPartialCollection` function, described in the previous section.

To create a flattened collection that includes every item in a collection, use the `FlattenCollection` function, described on page 5-88.

Working With Macintosh Memory Manager Handles

This section describes a set of utility functions provided by the Collection Manager that allow you to specify a collection item's variable-length data using a Macintosh Memory Manager handle.

AddCollectionItemHdl

You use the `AddCollectionItemHdl` function to add a new item to a collection or to replace an existing item in a collection, specifying the item's variable-length data using a handle rather than a pointer and a data size.

```
OSErr AddCollectionItemHdl (Collection target,
                            CollectionTag tag, long id,
                            Handle itemData);
```

`target`	A reference to the collection you want to add the item to.
`tag`	The collection tag you want to associate with the new item.
`id`	The collection ID you want to associate with the new item.
`itemData`	A Macintosh Memory Manager handle to the item's variable-length data.

DESCRIPTION

The `AddCollectionItemHdl` function adds an item to the collection referenced by the `target` parameter. This new item contains:

- the collection tag specified by the `tag` parameter
- the collection ID specified by the `id` parameter
- the attributes specified by the default attributes of the target collection
- the variable-length data specified by the `itemData` parameter

This function copies the information referenced by the `itemData` parameter into the new item; after calling this function, you may alter this information or free the memory referenced by this parameter without affecting the collection.

If the target collection already contains an item with the same collection tag and collection ID as specified in the `tag` and `id` parameters, this function removes the variable-length data from the original item and replaces it with the new data, unless the existing item is locked. If it is locked, this function returns a `collectionItemLockedErr` result code.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemLockedErr`	–5750	Can't replace locked item.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

For information about locking collection items, see "Getting and Setting the Attributes of an Item" beginning on page 5-24. To lock a collection item, use the functions described in "Editing Item Attributes" beginning on page 5-82.

To add or replace a collection item using a pointer (rather than a handle) to the item's variable-length data, use the `AddCollectionItem` function, described on page 5-62.

To replace a collection item using the item's collection index (rather than the item's collection tag and collection ID), use the `ReplaceIndexedCollectionItemHdl` function, described in the next section.

ReplaceIndexedCollectionItemHdl

You use the `ReplaceIndexedCollectionItemHdl` function to replace the variable-length data of an item in a collection given the item's collection index, specifying the item's new variable-length data using a handle rather than a pointer and a data size.

```
OSErr ReplaceIndexedCollectionItemHdl(Collection target,
                                      long index,
                                      Handle itemData);
```

`target`	A reference to the collection containing the item you want to replace.
`index`	The collection index associated with the item you want to replace.
`itemData`	A Macintosh Memory Manager handle to the new variable-length data.

DESCRIPTION

The `ReplaceIndexedCollectionItemHdl` function replaces the variable-length data of an item in the `target` collection. You specify which item to replace using the `index` parameter. If the target collection does not contain an item whose collection index matches the value of the `index` parameter, this function returns a `collectionIndexRangeErr` result code.

If the target collection does contain an item with the specified index, this function replaces the data in that item with new data (if the existing item is not locked—if it is, this function returns a `collectionItemLockedErr` result code). The resulting item contains

- the same collection tag as the original item
- the same collection ID as the original item
- the same attributes as the original item
- the variable-length data specified by the `itemData` parameter

This function copies the information referenced by the `itemData` parameter into the collection item; after calling this function, you may alter this information or free the memory referenced by this parameter without affecting the collection.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemLockedErr`	–5750	Can't replace locked item.
`collectionIndexRangeErr`	–5752	Index is out of range.

SEE ALSO

For information about collection items, see "Collection Items" beginning on page 5-8.

To replace a collection item using a pointer (rather than a handle) to the item's variable-length data, use the `ReplaceIndexedCollectionItem` function, described on page 5-63.

To replace a collection item using the item's collection tag and collection ID (rather than the item's collection index), use the `AddCollectionItemHdl` function, described in the previous section.

GetCollectionItemHdl

You can use the `GetCollectionItemHdl` function to obtain a copy of the variable-length data associated with a collection item given the item's collection tag and collection ID. You must provide a valid Macintosh Memory Manager handle for this function to copy the data into.

```
OSErr GetCollectionItemHdl(Collection source,
                           CollectionTag tag,
                           long id,
                           Handle itemData);
```

`source`	A reference to the collection object containing the item whose data you want to retrieve.
`tag`	The collection tag associated with the item whose data you want to retrieve.
`id`	The collection ID associated with the item whose data you want to retrieve.
`itemData`	A handle to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item. You may specify the constant `dontWantData` for this parameter if you do not want a copy of the item's data.

DESCRIPTION

The `GetCollectionItemHdl` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter and you specify an item in that collection using the `tag` and `id` parameters. If you provide a valid Macintosh Memory Manager handle in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8.

For examples using this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item into a block of memory referenced by a pointer (rather than a handle), use the `GetCollectionItem` function, described on page 5-71.

GetIndexedCollectionItemHdl

You can use the `GetIndexedCollectionItemHdl` function to copy the variable-length data associated with a collection item into a Macintosh Memory Manager handle, given the item's collection index.

```
OSErr GetIndexedCollectionItemHdl(Collection source,
                                  long index,
                                  Handle itemData);
```

`source` A reference to the collection object containing the item whose data you want to retrieve.

`index` The collection index associated with the item whose data you want to retrieve.

`itemData` A handle to a block of memory to contain the item's data. On return, this memory contains a copy of the data associated with the specified item.

DESCRIPTION

The `GetIndexedCollectionItemHdl` function allows you to obtain a copy of the variable-length data associated with a specific collection item. You specify a collection object using the `source` parameter and you specify an item in that collection using the `index` parameter. If you provide a valid Macintosh Memory Manager handle in the `itemData` parameter, the function uses this parameter to return a copy of the variable-length data associated with the specified collection item.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionItemNotFoundErr`	–5751	Can't locate item.

SEE ALSO

For information about collection items and their associated collection tags, collection IDs, and variable-length data, see "Collection Items" beginning on page 5-8.

For examples using this function, see "Retrieving the Variable-Length Data From an Item" beginning on page 5-33.

To retrieve the data associated with a collection item into a block of memory referenced by a pointer (rather than a handle), use the `GetCollectionItem` function, described on page 5-71.

FlattenCollectionToHdl

You use the `FlattenCollectionToHdl` utility function to flatten a collection into a Macintosh Memory Manager handle.

```
OSErr FlattenCollectionToHdl(Collection source
                             Handle flattened);
```

`source` The collection that you want to flatten into a handle.

`flattened` A handle to contain the flattened data.

DESCRIPTION

This function flattens the collection referenced by the `source` parameter into a block of memory referenced by the handle you provide in the `flattened` parameter.

You must provide a valid collection object reference in the `source` parameter and a valid Macintosh Memory Manager handle in the `flattened` parameter. You may specify a handle of size 0; this function resizes the handle as necessary to hold the flattened data.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.

SEE ALSO

For examples of this function, see "Reading Collections From and Writing Collections to Disk" beginning on page 5-41.

For an example that shows one possible implementation of this function, see "Flattening and Unflattening a Collection" beginning on page 5-37.

To flatten a collection directly to disk, use the `FlattenCollection` function, described on page 5-88.

To unflatten a collection from a block of memory referenced by a handle, use the `UnflattenCollectionFromHdl` function, described in the next section.

UnflattenCollectionFromHdl

You use the `UnflattenCollectionFromHdl` utility function to unflatten a collection that was flattened using the `FlattenCollectionToHdl` utility function.

```
OSErr UnflattenCollectionFromHdl(Collection target
                                 Handle flattened);
```

`target` A reference to a collection object in which to store the unflattened information.

`flattened` A handle to the data that was previously flattened.

DESCRIPTION

This function unflattens the information referenced by the handle you provide in the flattened parameter and stores the unflattened collection in the collection object referenced by the `target` parameter. You must provide a reference to a valid collection object in the `target` parameter and a valid Macintosh Memory Manager handle in the `flattened` parameter.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`collectionVersionErr`	–5753	Unrecognized version / data may be corrupt.

SEE ALSO

For examples of this function, see "Reading Collections From and Writing Collections to Disk" beginning on page 5-41.

For an example that shows one possible implementation of this function, see "Flattening and Unflattening a Collection" beginning on page 5-37.

To unflatten a collection directly from disk, use the `UnflattenCollection` function, described on page 5-90.

To flatten a collection to a block of memory referenced by a handle, use the `FlattenCollectionToHdl` function, described in the previous section.

Reading Collections From Resource Files

The function described in this section creates a collection object and initializes it with information stored in a `'cltn'` resource. You can find more information about `'cltn'` resources in "The Collection Resource" beginning on page 5-102.

You should be familiar with the information in the "Resource Manager" chapter of *Inside Macintosh: More Macintosh Toolbox* before using this function.

GetNewCollection

Use the `GetNewCollection` utility function to read a collection in from a collection (`'cltn'`) resource.

```
Collection GetNewCollection(short collectionID);
```

`collectionID`
The resource ID associated with the collection resource from which you want to create the new collection object.

function result A reference to the new collection object.

DESCRIPTION

This function searches the current resource file path for a collection (`'cltn'`) resource with the resource ID specified by the `collectionID` parameter. If it finds such a resource, this function creates a new collection object, initializes it with the information stored in the resource, and returns a reference to it as the function result.

If this function does not find a collection resource with the specified resource ID, it returns `nil` as the function result.

You can use the `MemError` and `ResError` functions to check for other errors after calling this function.

RESULT CODES

`memFullErr`	–108	Can't allocate memory.
`resNotFound`	–192	Resource not found.

SEE ALSO

For an example using this function, see "Reading a Collection From a Collection Resource" beginning on page 5-44.

For information about collection resources, see "The Collection Resource" beginning on page 5-102.

For more information about resources in general, see the "Resource Manager" chapter of *Inside Macintosh: More Macintosh Toolbox*.

Application-Defined Functions

This section describes two types of functions that you can provide to the Collection Manager:

- the callback function that you provide to the `FlattenCollection`, `FlattenPartialCollection`, and `UnflattenCollection` functions
- the exception procedure that you can provide for any collection object

MyFlattenProc

You provide the `MyFlattenProc` function to read or write flattened collection data.

```
OSErr MyFlattenProc(long size, void *data, void *refCon);
```

`size`	The size of the block of flattened data to read or write.
`data`	A pointer to the block of flattened data. When flattening, this pointer points to the data your callback function should write. When unflattening, your callback function should read flattened data into the memory pointed to by this parameter.
`refCon`	A value you provide to the `FlattenCollection` function or `UnflattenCollection` function that the Collection Manager passes on to your callback function.

DESCRIPTION

You create this function to pass to the `FlattenCollection`, `FlattenPartialCollection`, and `UnflattenCollection` functions when flattening or unflattening a collection.

As the Collection Manager is flattening a collection, it repeatedly calls this callback function to process sequential blocks of flattened data. Each time it calls this function, it provides a pointer to the current block of flattened data in the `data` parameter and the size of the current block in the `size` parameter. You can process this data in a number of ways: appending it to a handle-based block of memory, writing it to disk, and so on.

When unflattening a collection, the Collection Manager repeatedly calls this function to obtain blocks of flattened data. The Collection Manager specifies the size of the requested block in the `size` parameter, and your function should read or copy the requested number of bytes of flattened data into the block of memory pointed to by the `data` parameter.

In either case, the Collection Manager passes in the `refCon` parameter the same value you originally passed as the `refCon` parameter to the `FlattenCollection`, `FlattenPartialCollection`, or `UnflattenCollection` function. You can use this parameter as a pointer to a structure containing relevant state information you need when reading or writing the flattened data.

If the execution of this function results in any fatal error, you should return the error code back to the Collection Manager as the function result. If the function executes successfully, you should return the `noErr` error code as the function result.

SEE ALSO

For more information about the flattening and unflattening functions, see "Flattening and Unflattening a Collection" beginning on page 5-88.

For examples of this function, see "Flattening and Unflattening a Collection" beginning on page 5-37.

MyExceptionProc

You provide the `MyExceptionProc` function (an exception procedure) to handle errors that occur when operating on a collection object.

```
OSErr MyExceptionProc(Collection target, OSErr whichErr);
```

`target`	A reference to the collection object for which the error occurred.
`whichErr`	The result code associated with the error that occurred.

DESCRIPTION

You create this function to install in a collection object using the `SetCollectionExceptionProc` function. Subsequently, whenever the Collection Manager is operating on that collection object and an error occurs, the Collection Manager calls this function, sending it a reference to the collection for which the error occurred and the result code associated with the error. You can use this information to handle the error appropriately for your application.

You can use an exception procedure to respond to an error in a number of ways:

- You can change the error from one result code to another by returning as the function result the new result code.
- You can handle the error and return the `noErr` error code, which indicates that the Collection Manager should return control to the place in your application that generated the error, as if no error had occurred.
- You can use the ANSI C functions `setjmp` and `longjmp` to jump out of the exception procedure into code to handle the error.

SEE ALSO

For an example of an exception procedure see “Installing an Exception Procedure” beginning on page 5-45.

To install an exception procedure in a collection object, use the `SetCollectionExceptionProc` function, which is described on page 5-59.

To obtain a pointer to an existing exception procedure in a collection object, use the `GetCollectionExceptionProc` function, which is described on page 5-58.

Resources

This section describes the structure of the collection resource and the meaning of its fields.

The Collection Resource

The Collection Manager provides the `GetNewCollection` function, described on page 5-99, to create a new collection object and initialize it using information stored in a collection (`'cltn'`) resource. Listing 5-28 shows the structure of the collection resource in Rez format.

Listing 5-28 A Rez template for a `'cltn'` resource

```
type 'cltn' {
   longint = $$CountOf(ItemArray);
   array ItemArray
      {
      longint; /* tag */
      longint; /* id */
         boolean  itemUnlocked = false, /* defined attributes */
                  itemLocked = true;
         boolean  itemNonPersistent = false,
                  itemPersistent = true;
         unsigned bitstring[14] = 0; /* reserved attributes */
         unsigned bitstring[16] userBits; /* user attributes */
      wstring;
      align word;
   };
};
```

The collection resource has two parts:

- a count of the number of items in the resource
- an array of items

Each item in the array specifies

- the collection tag for that item
- the collection ID for the item
- a Boolean value representing the lock attribute for the item
- a Boolean value representing the persistence attribute for the item
- 14 bits representing the 14 reserved attributes for the item
- 16 bits representing the 16 user-defined attributes for the item
- a string containing the variable-length data for the item

Summary of the Collection Manager

Data Types

Optional Return Value Constants

```
enum {
   dontWantTag          = 0L,  /* don't want collection tag returned */
   dontWantId           = 0L,  /* don't want collection ID returned */
   dontWantSize         = 0L,  /* don't want size of data returned */
   dontWantAttributes   = 0L,  /* don't want attributes returned */
   dontWantIndex        = 0L,  /* don't want collection index returned */
   dontWantData         = 0L   /* don't want variable-length data returned */
};
```

Attributes Masks

```
enum {
   noCollectionAttributes      = 0x00000000,  /* no attributes bits set */
   allCollectionAttributes     = 0xFFFFFFFF,  /* all attributes bits set */
   userCollectionAttributes    = 0x0000FFFF,  /* user attributes bits set */
   defaultCollectionAttributes = 0x40000000   /* unlocked, persistent */
};
```

Attribute Bit Numbers

```
enum {
   collectionUser0Bit   = 0,  /* for use by application */
   collectionUser1Bit   = 1,
   collectionUser2Bit   = 2,
   collectionUser3Bit   = 3,
   collectionUser4Bit   = 4,
   collectionUser5Bit   = 5,
   collectionUser6Bit   = 6,
   collectionUser7Bit   = 7,
   collectionUser8Bit   = 8,
   collectionUser9Bit   = 9,
   collectionUser10Bit  = 10,
   collectionUser11Bit  = 11,
```

```
   collectionUser12Bit  = 12,
   collectionUser13Bit  = 13,
   collectionUser14Bit  = 14,
   collectionUser15Bit  = 15,

   collectionReserved0Bit  = 16,  /* reserved for use by Apple */
   collectionReserved1Bit  = 17,
   collectionReserved2Bit  = 18,
   collectionReserved3Bit  = 19,
   collectionReserved4Bit  = 20,
   collectionReserved5Bit  = 21,
   collectionReserved6Bit  = 22,
   collectionReserved7Bit  = 23,
   collectionReserved8Bit  = 24,
   collectionReserved9Bit  = 25,
   collectionReserved10Bit = 26,
   collectionReserved11Bit = 27,
   collectionReserved12Bit = 28,
   collectionReserved13Bit = 29,

   collectionPersistenceBit = 30,  /* currently defined by Apple */
   collectionLockBit        = 31
};
```

Attribute Bit Masks

```
enum {
   collectionUser0Mask = 1L << collectionUser0Bit,
   collectionUser1Mask = 1L << collectionUser1Bit,
   collectionUser2Mask = 1L << collectionUser2Bit,
   collectionUser3Mask = 1L << collectionUser3Bit,
   collectionUser4Mask = 1L << collectionUser4Bit,
   collectionUser5Mask = 1L << collectionUser5Bit,
   collectionUser6Mask = 1L << collectionUser6Bit,
   collectionUser7Mask = 1L << collectionUser7Bit,
   collectionUser8Mask = 1L << collectionUser8Bit,
   collectionUser9Mask = 1L << collectionUser9Bit,
   collectionUser10Mask = 1L << collectionUser10Bit,
   collectionUser11Mask = 1L << collectionUser11Bit,
   collectionUser12Mask = 1L << collectionUser12Bit,
   collectionUser13Mask = 1L << collectionUser13Bit,
   collectionUser14Mask = 1L << collectionUser14Bit,
   collectionUser15Mask = 1L << collectionUser15Bit,
```

```
   collectionReserved0Mask = 1L << collectionReserved0Bit,
   collectionReserved1Mask = 1L << collectionReserved1Bit,
   collectionReserved2Mask = 1L << collectionReserved2Bit,
   collectionReserved3Mask = 1L << collectionReserved3Bit,
   collectionReserved4Mask = 1L << collectionReserved4Bit,
   collectionReserved5Mask = 1L << collectionReserved5Bit,
   collectionReserved6Mask = 1L << collectionReserved6Bit,
   collectionReserved7Mask = 1L << collectionReserved7Bit,
   collectionReserved8Mask = 1L << collectionReserved8Bit,
   collectionReserved9Mask = 1L << collectionReserved9Bit,
   collectionReserved10Mask = 1L << collectionReserved10Bit,
   collectionReserved11Mask = 1L << collectionReserved11Bit,
   collectionReserved12Mask = 1L << collectionReserved12Bit,
   collectionReserved13Mask = 1L << collectionReserved13Bit,

   collectionPersistenceMask = 1L << collectionPersistenceBit,
   collectionLockMask = 1L << collectionLockBit
};
```

Functions

Creating and Disposing of Collection Objects

```
Collection NewCollection    (void);
void DisposeCollection      (Collection target);
```

Cloning and Copying Collection Objects

```
Collection CloneCollection  (Collection target);
long CountCollectionOwners  (Collection source);
Collection CopyCollection   (Collection source,
                             Collection target);
```

Getting and Setting the Exception Procedure for a Collection

```
CollectionExceptionProc GetCollectionExceptionProc
                            (Collection source);
void SetCollectionExceptionProc
                            (Collection target,
                             CollectionExceptionProc newExceptionProc);
```

Getting and Setting the Default Attributes for a Collection

```
long GetCollectionDefaultAttributes
                            (Collection source);
void SetCollectionDefaultAttributes
                            (Collection target,
                             long whichAttributes,
                             long newAttributes);
```

Adding and Replacing Items in a Collection

```
OSErr AddCollectionItem     (Collection target,
                             CollectionTag tag, long id,
                             long itemSize, void *itemData);
OSErr ReplaceIndexedCollectionItem
                            (Collection target, long index,
                             long itemSize, void *itemData);
```

Removing Items From a Collection

```
OSErr RemoveCollectionItem  (Collection target,
                             CollectionTag tag, long id);
OSErr RemoveIndexedCollectionItem
                            (Collection target, long index);
void PurgeCollection        (Collection target,
                             long whichAttributes,
                             long matchingAttributes);
void PurgeCollectionTag     (Collection target, CollectionTag tag);
void EmptyCollection        (Collection target);
```

Counting Items in a Collection

```
long CountCollectionItems   (Collection source);
long CountTaggedCollectionItems
                            (Collection source, CollectionTag tag);
```

Retrieving the Variable-Length Data From an Item

```
OSErr GetCollectionItem     (Collection source,
                             CollectionTag tag, long id,
                             long *itemSize, void *itemData);
OSErr GetIndexedCollectionItem
                            (Collection source, long index,
                             long *itemSize, void *itemData);
OSErr GetTaggedCollectionItem
                            (Collection source,
                             CollectionTag tag, long position,
                             long *itemSize, void *itemData);
```

Getting Information About a Collection Item

```
OSErr GetCollectionItemInfo (Collection source,
                             CollectionTag tag, long id,
                             long *index, long *itemSize,
                             long *attributes);
OSErr GetIndexedCollectionItemInfo
                            (Collection source, long index,
                             CollectionTag *tag, long *id,
                             long *itemSize, long *attributes);
OSErr GetTaggedCollectionItemInfo
                            (Collection source,
                             CollectionTag tag, long position,
                             long *id, long *index,
                             long *itemSize, void *attributes);
```

Editing Item Attributes

```
OSErr SetCollectionItemInfo
                            (Collection target,
                             CollectionTag tag, long id,
                             long whichAttributes, long newAttributes);
OSErr SetIndexedCollectionItemInfo
                            (Collection target, long index,
                             long whichAttributes, long newAttributes);
```

Getting Information About Collection Tags

```
Boolean CollectionTagExists
                            (Collection source, CollectionTag tag);
long CountCollectionTags    (Collection source);
OSErr GetIndexedCollectionTag
                            (Collection source, long whichTag,
                             CollectionTag *tag);
```

Flattening and Unflattening a Collection

```
OSErr FlattenCollection     (Collection source,
                             CollectionFlattenProc flattenProc,
                             void *refCon);
OSErr FlattenPartialCollection
                            (Collection source,
                             CollectionFlattenProc flattenProc,
                             void *refCon,
                             long whichAttributes,
                             long matchingAttributes);
OSErr UnflattenCollection   (Collection target,
                             CollectionFlattenProc flattenProc,
                             void *refCon);
```

Working With Macintosh Memory Manager Handles

```
OSErr AddCollectionItemHdl  (Collection target,
                             CollectionTag tag, long id,
                             Handle itemData);
OSErr ReplaceIndexedCollectionItemHdl
                            (Collection target, long index,
                             Handle itemData);
OSErr GetCollectionItemHdl  (Collection source,
                             CollectionTag tag, long id,
                             Handle itemData);
OSErr GetIndexedCollectionItemHdl
                            (Collection source, long index,
                             Handle itemData);
OSErr FlattenCollectionToHdl
                            (Collection source, Handle flattened);
OSErr UnflattenCollectionFromHdl
                            (Collection target, Handle flattened);
```

Reading Collections From Resource Files

```
Collection GetNewCollection (short collectionID);
```

Application-Defined Functions

```
OSErr MyFlattenProc         (long size, void *data, void *refCon);
OSErr MyExceptionProc       (Collection target, OSErr whichErr);
```

Resources

The Collection Resource

```
type 'cltn' {
   longint = $$CountOf(ItemArray);
   array ItemArray
      {
      longint; /* tag */
      longint; /* id */
         boolean  itemUnlocked = false, /* defined attributes */
                  itemLocked = true;
         boolean  itemNonPersistent = false,
                  itemPersistent = true;
         unsigned bitstring[14] = 0; /* reserved attributes */
         unsigned bitstring[16] userBits; /* user attributes */
      wstring;
      align word;
   };
};
```