HIArchive Programming Guide

Carbon > Human Interface Toolbox

2005-08-11

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to HIArchive Programming Guide

HIArchive provides a convenient and standardized mechanism for flattening data objects so they can be stored in memory or on disk. Applications can use these archives whenever they need to package complex data. For example, you can use archives to:

- Store document data
- Transfer data using pasteboards, drag and drop, streams, or Apple events
- Store localization strings and user interface elements in the same package

HIArchive encodes archives in the binary property list format. You can convert archives to a text XML format using the `plutil` property list tool accessible from Terminal. You can also examine archives using the Property List Editor tool in `/Developer/Applications/Utilities`.

Who Should Read This Document?

This document is for Carbon developers who want to use, create, or manipulate HIArchives, whether to store and access proprietary data, or to edit archived data obtained from other sources. You should also read this document if you want to support the archiving of your custom HIObjects.

HIArchive is comparable to (and uses the same underlying mechanism as) the Cocoa NSKeyedArchiver/Unarchiver classes.

HIArchive is available in Mac OS X version 10.4 and later.

Organization of This Document

This document is organized into the following chapters:

- "Archiving and Unarchiving Objects" (page 9) describes the basics of using HIArchives.
- "Making HIObjects Archivable" (page 15) describes how to make your custom HIObjects support archiving.

See Also

In addition to this document, you may find the following documents useful:

- For a complete description of the HIArchive API, see *HIArchive Reference*.
- If you are not familiar with using HIViews and HIObjects, you should read *HIView Programming Guide*.

Archiving and Unarchiving Objects

HIArchive provides a convenient way to store data objects in a portable format. This chapter describes the basics of archiving and unarchiving objects using HIArchive APIs.

What Can Be Archived?

You can use HIArchive to archive any CFPropertyList data types. Some examples:

- CFArray
- CFData
- CFString
- CFDictionary
- CFDate
- CFBoolean
- CFNumber

CFPropertyList collection types are archivable if they contain only archivable objects.

You can also archive any HIObject that supports the HIArchive protocol. All the standard HIViews (menus, controls) and windows support archiving. If you use custom views, you need to add some additional code to support archiving. See "Making HIObjects Archivable" (page 15) for details.

In addition, you can archive other CFTypes by manually serializing them to CFData objects (which are archivable).

All data is stored in the HIArchive as key value pairs.

Using Default Values for Efficiency

Often when archiving data, you may find that certain item values are unchanged from their initial or default values. For example, a custom view may have bounds that, while modifiable, are more often left in their initial state. In cases where you would encode known default values into an archive, you can leave such items out. Then during decoding, if an expected key does not exist, you should assign that item its default value. Doing so minimizes archive space and encoding/decoding time.

However, keep the following thoughts in mind:

- If you choose to not write a key value pair to the archive if the object data has the default value, make sure you don't change the default value in a future version of your software.

- You should not change the meaning of a key, as this could cause problems for older software unarchiving newer objects. If you feel you need to change a key, consider using a new one instead, and write both keys to the archive. Older software can read the old key. Newer software can read the new key, if present, or the old key if not.

The examples in this document check for default values and do not write them to the archive.

Archiving Objects

To write data to an archive, your application must first create a write-only archive (specified by an `HIArchiveRef` object) by calling `HIArchiveCreateForEncoding`.

To add data to the archive, you call the appropriate encoding function:

- `HIArchiveEncodeBoolean` for Boolean values
- `HIArchiveEncodeNumber` for any numerical values
- `HIArchiveEncodeCFType` for any CFPropertyList types. You also use this function to encode HIObjects or their subclasses.

Note: The `HIArchiveEncodeBoolean` and `HIArchiveEncodeNumber` functions are wrappers that call `HIArchiveEncodeCFType` with the appropriate CFBoolean or CFNumber value.

All data is encoded with a (CFStringRef) key, which uniquely identifies the data within the archive. The keys must be unique only within the current object you are encoding. For example, keys used by object A do not conflict with keys used by object B, even if A and B are instances of the same class. Within a single object, however, keys used by a subclass can conflict with keys used by its superclass. If you overwrite a superclass key, HIArchive warns you by sending a message to the console output; it is up to you to decide whether this message indicates an error.

System-supplied HIObjects always have an `HI` prefix in the key name; your custom HIObject subclasses should avoid using this prefix unless you are explicitly overriding a value written to the archive by the superclass. With careful use of keys, your archives can support versioning; on older versions, newly keyed data written on a more recent version of software or OS is ignored.

If you call `HIArchiveEncodeCFType` on your own custom HIObject, the system sends a `kEventHIObjectEncode` Carbon event to the object. It is then your custom object's responsibility to encode the appropriate instance data into the archive specified in the `kEventParamHIArchive` parameter using the `HIArchiveEncodeXXX` calls. To receive the `kEventHIObjectEncode` event, your HIObject must indicate that it supports archiving by passing `false` to the HIObject function `HIObjectSetArchivingIgnored`. For more details, see "Making HIObjects Archivable" (page 15).

After you have encoded all the data into the archive, call `HIArchiveCopyEncodedData` to compress the data. After compression, you handle the archive using the returned Core Foundation data reference (`CFDataRef`). You can use this reference to write the archive to disk, pass it to another application, copy it to a pasteboard, and so on. After compression, you can no longer write to the archive, and you must release the original archive reference (`HIArchiveRef`) by calling `CFRelease`.

Listing 1-1 (page 11) shows how you might encode data items into an archive and then write the archive to a file specified by a URL.

Listing 1-1 Encoding items and writing to a file

```
#define kFirstItemKey CFSTR("myFirstItemKey");
#define kSecondItemKey CFSTR("mySecondItemKey");

OSStatus ArchiveObjectsInFile (CFTypeRef firstItem,
                               CFTypeRef secondItem, CFURLRef inFileURL )
{
    OSStatus err = noErr;
    HIArchiveRef encoder;
    CFDataRef encodedData;

    err = HIArchiveCreateForEncoding( &encoder );                              // 1
    require_noerr (err, cantCreateEncoder);

    if (!CFEqual( firstItem, kDefaultFirstItem))                               // 2
    {
        err = HIArchiveEncodeCFType( encoder, kFirstItemKey, firstItem );      // 3
        require_noerr (err, cantEncodeObject);
    }

    if (!CFEqual( secondItem, kDefaultSecondItem))                             // 4
    {
        err = HIArchiveEncodeCFType( encoder, kSecondItemKey, secondItem );
        require_noerr (err, cantEncodeObject);
    }

    err = HIArchiveCopyEncodedData( encoder, &encodedData );                   // 5
    require_noerr (err, cantCopyEncodedData);

    verify(
        CFURLWriteDataAndPropertiesToResource( inFileURL, encodedData,         // 6
                               NULL, NULL ));

    CFRelease ( encodedData );                                                 // 7

cantEncodeObject:
cantCopyEncodedData:

    CFRelease ( encoder );                                                     // 8

cantCreateEncoder:

    return err;
}
```

Here is how the code works:

1. Creates an HIArchive to hold the encoded data.

2. Checks to see if the value to be archived is the same as the default value. The default value could be any value you would commonly expect to see for this archivable item; an initial position or setting, standard size, default attribute, and so on. If the value to be archived is the same as the default, you can skip the archiving procedure, which saves space and processing time. Of course, you must make sure that your unarchiving function automatically inserts default values for items that do not appear in the archive.
3. Calls the `HIArchiveEncodeCFType` function to add the item to the archive by key.
4. Repeats the archiving process for the second item.
5. After encoding all the items, calls `HIArchiveCopyEncodedData` to flatten the archived items into a CFData object.
6. Writes the CFData object to a file URL.
7. Releases the CFData object.
8. Releases the archive.

Unarchiving Objects

To read data from an archive, your application must create a read-only archive from the specified `CFDataRef` by calling `HIArchiveCreateForDecoding`. You retrieve data from the archive using the appropriate decoding functions:

- `HIArchiveDecodeBoolean`
- `HIArchiveDecodeNumber`
- `HIArchiveCopyDecodedCFType`

Note: Again, the `HIArchiveDecodeBoolean` and `HIArchiveDecodeNumber` functions are wrappers that call `HIArchiveCopyDecodedCFType` with the appropriate CFBoolean or CFNumber value.

If you call `HIArchiveCopyDecodedCFType` to retrieve a custom HIObject from an archive, the system sends a `kEventHIObjectInitialize` event to the object. Your HIObject's initialization handler must then retrieve data for its custom HIObject from the `kEventParamHIArchive` parameter using the `HIArchiveDecodeXXX` calls. See see "Making HIObjects Archivable" (page 15) for details.

When you finish retrieving data from the archive, call `CFRelease` to release the archive reference.

Listing 1-2 (page 12) shows how you might unarchive data using a CFData reference. This data may be the reference obtained from a `HIArchiveCopyEncodedData` call or a copy obtained from a file, URL, or other source. For example, you could call `CFURLCreateDataAndPropertiesFromResource`, to load the XML data from an arbitrary URL.

Listing 1-2 Decoding items from a CFData reference

```
OSStatus LoadObjectsFromCFData( CFTypeRef* firstItem,
                            CFTypeRef* secondItem, CFDataRef inData )
{
```

```
    OSStatus err = noErr;
    HIArchiveRef decoder;

    err = HIArchiveCreateForDecoding( inData, 0, &decoder );                    // 1
    require_noerr( err, cantCreateDecoder );

    err = HIArchiveCopyDecodedCFType( decoder, kFirstItemKey, firstItem);       // 2

    if (err == hiArchiveKeyNotAvailableErr)                                     // 3
        *firstItem = CFRetain( kDefaultFirstItem);
    else
        require_noerr( err, cantDecodeObjectFromData );

    err = HIArchiveCopyDecodedCFType( decoder, kSecondItemKey, secondItem );    // 4

    if (err == hiArchiveKeyNotAvailableErr)
        *secondItem = CFRetain( kDefaultSecondItem);
    else
        require_noerr( err, cantDecodeObjectFromData );

cantDecodeObjectFromData:

    CFRelease( decoder );                                                       // 5

cantCreateDecoder:

    return err;
}
```

Here is how the code works:

1. Creates an HIArchive for decoding items.
2. Attempts to decode the first archive object by key name.
3. Assigns a default value for this item if the error indicates that the specified key does not exist in the archive . If you are opening an older archive that does not contain the latest items, you can also use defaults to populate the missing values.
4. Repeats the unarchiving for the second item.
5. Releases the archive.

Editing Archives

If you want to create a generic HIArchive editor, you should keep the following in mind:

- Because the editor does not have any prior knowledge of what keys and data exist, you may want to obtain the archive as a CFPropertyList, which you can then parse to obtain key names. You can do so by calling the Core Foundation function `CFPropertyListCreateFromXMLData`.

- A generic editor will probably encounter archives containing custom HIObject subclasses that have not been registered with the system. In such cases, you should make sure to specify the `kHIArchiveDecodeSuperClassForUnregisteredObject` option when calling `HIArchiveCreateForDecoding`. When HIArchive encounters an unregistered subclass, it instantiates its superclass instead and attaches any custom data to that object. The custom data is comparable to the information available in the Attributes pane of the Inspector window for custom HIViews in Interface Builder. You can obtain the custom (that is, subclass-specific) data by calling the HIObject function `HIObjectCopyCustomArchiveData`.You receive the data as a CFDictionary with keys defined in `HIObject.h`. See HIObject Reference for more details.
- When writing an unregistered subclass object to an archive, your editor must call the HIObject function `HIObjectSetCustomArchiveData`, passing a CFDictionary containing subclass-specific data. You should write the dictionary data as key value pairs using the dictionary keys supplied in `HIObject.h` (specifying, for example, initialization parameters, and class and superclass identifiers).

Making HIObjects Archivable

If you want your custom HIObjects (such as custom HIViews) to support HIArchiving, you need to implement some additional code to do so. This chapter describes the modifications you need to make.

How to Support HIArchive Encoding

To support HIArchive encoding, your HIObject must be able to respond to the `kEventHIObjectEncode` event. Your custom HIObject receives this event during encoding, and it is the responsibility of the HIObject to encode its instance data into the provided HIArchive.

You encode your HIObject instance data just as you would any other data, using the `HIArchiveEncodeBoolean`, `HIArchiveEncodeNumber`, or `HIArchiveEncodeCFType` functions, giving each value a unique key.

In addition to supporting the `kEventHIObjectEncode` event, your HIObject must also indicate that it supports archiving by passing `false` to the HIObject function `HIObjectSetArchivingIgnored`. Typically you do so in your HIObject's `kEventHIObjectInitialize` event handler. If you don't call this function, your HIObject will never receive the encoding event.

Listing 1-1 (page 15) shows how you can implement the handler for the `kEventHIObjectEncode` event.

Listing 2-1 An `kEventHIObjectEncode` event handler

```
OSStatus MyHIObjectEncode(
  EventHandlerCallRef inCallRef,
  EventRef inEvent,
  void* inRefCon )
{
 OSStatus err;
 HIArchiveRef encoder;
 MyHIViewData* myData = (MyHIViewData*)inRefCon;

 err = CallNextEventHandler( inCallRef, inEvent );                          // 1
 require_noerr( err, cantEncodeSuperclass );

 err = GetEventParameter( inEvent, kEventParamHIArchive, typeCFTypeRef,     // 2
                          NULL, sizeof( HIArchiveRef ), NULL, &encoder );
 require_noerr (err, cantGetArchive);

 if (!CFEqual (myData->myFirstDataItem, kDefaultFirstItemValue))            // 3
 {
    err = HIArchiveEncodeCFType( encoder, kMyFirstDataItemArchiveKey,       // 4
                                 myData-> myFirstDataItem );
    require_noerr( err, cantEncodeItem );
 }
```

```
 if (!CFEqual( myData->mySecondDataItem, kDefaultSecondItemValue))          // 5
 {
    err = HIArchiveEncodeNumber ( encoder, kMySecondDataItemArchiveKey,
                    kCFNumberCFIndexType, &(myData->mySecondDataItem ));
    require_noerr( err, cantEncodeItem );
 }

cantEncodeItem:
cantEncodeSuperclass:
cantGetArchive:

  return err;
}
```

Here is how the code works:

1. As usual, your event handler must call the Carbon Event Manager function `CallNextEventHandler` to allow the superclass a chance to encode its data into the archive. If you are subclassing from HIView, the HIView base class will archive basic HIView data such as the view's size, bounds, and so on.

2. Obtains the HIArchive reference to encode into. This reference is packaged in the `kEventHIObjectEncode` event.

3. Checks to see if the value of the first instance data item is the same as some default value. If so, you don't need to encode this item (as long as your decoder knows to assign the default value for any nonexistent keys) Doing so saves space in the archive (and minimizes processing time).

4. Encodes the first instance data item. HIView. You must call `HIArchiveEncodeCFType` or one of its wrapper variants (for CFBooleans or CFNumbers) for each field in your instance data structure. In this example, the instance data structure would look something like this:

   ```
   typedef struct
   {
       HIViewRef view;
       CFStringRef myFirstDataItem;
       CFIndex mySecondDataItem;
   } MyHIViewData;
   ```

   Notice that you don't have to encode the HIView reference (`HIViewRef`), because HIArchive does this for you automatically.

   The archive keys (for example, `kMyFirstDataItemArchiveKey`), are application-defined CFString constants that uniquely identify each data item you want to archive (and later retrieve).

5. Repeat for the second instance data item.

How to Support HIArchive Decoding

To support HIArchive decoding, your HIObject must be able to instantiate itself from archive data. Doing so requires that your `kEventHIObjectInitialize` event handler be able to extract instance data for your object from an HIArchive.

When your custom HIObject receives the `kEventHIObjectInitialize` event, you could check to see if an HIArchive parameter was passed to you. If so, you should unarchive the instance data before proceeding with any standard initalization.

Listing 1-2 (page 17) shows how to decode archive information within your `kEventHIObjectInitialize` event handler.

Listing 2-2 Decoding items in a `kEventHIObjectInitialize` event handler

```
OSStatus MyHIObjectInitialize(
    EventHandlerCallRef inCallRef,
    EventRef inEvent,
    void* inRefCon )
{
    OSStatus err = noErr;
    HIArchiveRef decoder = NULL;
    MyHIViewData* myData = (MyHIViewData*)inRefCon;

    err = CallNextEventHandler( inCallRef, inEvent );
    require_noerr( err, cantInitializeSuperclass );

    GetEventParameter( inEvent, kEventParamHIArchive, typeCFTypeRef, NULL,     // 1
                       sizeof( HIArchiveRef ), NULL, &decoder );

    if ( decoder != NULL )                                                     // 2
        err = HIArchiveCopyDecodedCFType( decoder,
        kMyFirstDataItemArchiveKey, (CFTypeRef*)&myData->myFirstDataItem );

    if (decoder == NULL || err == hiArchiveKeyNotAvailableErr)                 // 3
        myData->myFirstDataItem = CFRetain ( kDefaultFirstItem );

    err = HIArchiveDecodeNumber( decoder, kMySecondDataItemArchiveKey,         // 4
                    kCFNumberCFIndexType, &(myData->mySecondDataItem ));

    if (decoder == NULL || err == hiArchiveKeyNotAvailableErr)
        myData->mySecondDataItem = CFRetain( kDefaultSecondItem );

    //perform any common initialization here

    HIObjectSetArchivingIgnored ( myData->view, false );                       // 5

cantInitializeSuperclass:

    return err;
}
```

Here is how the code works:

1. Attempts to obtain the HIArchive parameter. If the initialization is occurring in response to an unarchiving attempt, HIArchive automatically supplies the appropriate HIArchive reference in the initialization event.
2. Attempts to decode the first archive object by key name.
3. If the decoder did not exist (that is, this is a standard initialization) or if the key did not exist, sets the first item to its default value.
4. Repeat for the second item.

5. Sets the archiving ignored attribute to `false` to indicate that this HIObject supports the HIArchive protocol. Doing so ensures that the HIObject receives `kEventHIObjectEncode` events.

Adding Additional Archivable Information

Sometimes you want to associate additional information with an HIObject before you archive it. For example, you may want to store initialization parameters with the HIObject, or metadata that is useful to your application. In most cases, this archive data is useful only if you are writing an HIArchive editor.

Use the HIObject function `HIObjectSetCustomArchiveData` to associate a CFDictionary with an HIObject:

```
OSStatus HIObjectSetCustomArchiveData (
   HIObjectRef inObject,
   CFDictionaryRef inCustomData
);
```

When you set this archive information, the dictionary is automatically archived by the HIObject base class when it receives the `kEventHIObjectEncode` event. (Remember to call `CallNextEventHandler` if you handle the encoding event.)

To retrieve archived data, you use the `HIObjectCopyCustomArchiveData` function:

```
OSStatus HIObjectCopyCustomArchiveData (
   HIObjectRef inObject,
   CFDictionaryRef* outCustomData
);
```

HIObject defines several keys to use when adding standard data (such as initialization parameters or class IDs) to an archive dictionary. You can also define your own keys if necessary. For example, you use the following keys to store initialization parameters:

```
const CFStringRef kHIObjectCustomDataParameterNamesKey;
const CFStringRef kHIObjectCustomDataParameterTypesKey;
const CFStringRef kHIObjectCustomDataParameterValuesKey;
```

Each key represents a CFArray of initialization parameter names, types, or values. These keys correspond to the initialization parameters you can set for a custom HIView in Interface Builder (in the Attributes pane of the Inspector window).

Internally, when unarchiving your custom HIObject, HIArchive automatically extracts any initialization parameter information from the archive data, packages that in a `kEventHIObjectInitialize` event, and sends the event to your object. If for some reason HIArchive chooses to instantiate your HIObject superclass instead (your HIObject class was not registered), you can still access the initialization parameters through this archive dictionary. An HIArchive editor can obtain the data in this manner so that the user can view or change it.

You can also store the HIObject class and superclass IDs:

```
const CFStringRef kHIObjectCustomDataClassIDKey;
const CFStringRef kHIObjectCustomDataSuperClassIDKey;
```

Note that HIArchive automatically stores the class ID of a custom HIObject during the archiving process. As a result, you need to use the class and superclass keys only if you are editing an existing archive.

For more information about custom archiving keys, see *HIObject Reference*.

Document Revision History

This table describes the changes to *HIArchive Programming Guide*.

Date	Notes
2005-08-11	New document describing how to store data objects using HIArchive. Also includes information to make custom HIObjects archivable.

Document Revision History