Pasteboard Manager Programming Guide

Carbon > Interapplication Communication

2005-07-07

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Pasteboard Manager Programming Guide

Pasteboards are the standard data interchange mechanism for applications on Mac OS X, supported in both Carbon and Cocoa. The Pasteboard Manager is the Carbon programming interface for creating and accessing pasteboards.

Who Should Read This Document

This document is for Carbon developers who want to use pasteboards in their applications. The most common uses for the pasteboard are:

- to enable copy-and-paste actions using the Clipboard
- to implement drag-and-drop behavior
- to copy or retrieve text in the standard search field
- to transfer data to and from Mac OS X services (using either the Services menu or Translation Services)

You can also use pasteboards for any other purpose, such as implementing a proprietary clipboard. Pasteboard Manager pasteboards are fully-compatible with Cocoa NSPasteboard objects.

The Pasteboard Manager is available in Mac OS X v10.3 and later.

The Pasteboard Manager replaces both the older Scrap Manager and the Drag Manager's drag flavor APIs. While the Scrap Manager is still supported, the Pasteboard Manager provides greater flexibility and functionality.

Organization of This Document

This document is organized into the following chapters:

- "Pasteboard Manager Concepts" (page 9) describes pasteboard terminology and concepts.
- "Pasteboard Manager Tasks" (page 13) describes how to use the Pasteboard Manager in common tasks, such as cut-and-paste and drag-and-drop.
- "Scrap Manager Versus the Pasteboard Manager" (page 31) specifies replacement APIs for Scrap Manager functions.

See Also

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

- If you are not familar with uniform type identifiers (UTIs), you should read *Uniform Type Identifiers Overview*.
- If you are not already familiar with Carbon events, you should read *Carbon Event Manager Programming Guide*.
- If you are not familiar with using HIViews, you should read *HIView Programming Guide*.
- For more information about implementing services, see *Setting Up Your Carbon Application to Use the Services Menu*.

Pasteboard Manager Concepts

This section describes the terminology and concepts that underlie pasteboards.

What's a Pasteboard?

A pasteboard is a standardized mechanism for exchanging data within applications or between applications. The most familiar use for pasteboards is handling copy and paste operations. When a user selects data in an application and chooses the Copy (or Cut) menu item, the selected data is placed into the Clipboard pasteboard. When the user chooses the Paste menu item (either in the same or a different application), the data in the Clipboard is copied to the current application.

A pasteboard is analogous to a message board or bulletin board, which allows various people to exchange information asynchronously. Some bulletin boards, such as one in a household kitchen, may be accessible only to immediate family members. Others may be available to the public. Similarly, pasteboards may be public or private, and may be used for a variety of purposes.

Pasteboards exist in a special global memory area separate from application processes. Two special pasteboards exist:

- Clipboard. This is the pasteboard used to handle all copy-and-paste operations.
- Find. This is the pasteboard used to hold the current search string in Find operations.

In addition, the Drag Manager and Services Manager use pasteboards:

- When a user begins a drag, the drag data is added to a pasteboard. If the drag ends with a drop action, the receiving application retrieves the drag data from the pasteboard.
- If a translation service is requested, the requesting application places the data to be translated onto a pasteboard. The service retrieves this data, performs the translation, and places the translated data back onto the pasteboard.

Pasteboard Manager Versus the Scrap Manager

The Pasteboard Manager is a more modern and robust replacement for the original Scrap Manager. While the Scrap Manager is currently still supported, you should update your applications to use the Pasteboard Manager instead. Some of the advantages of the Pasteboard Manager include:

- The ability to store multiple items at a time.
- More advanced flavor typing using uniform type identifiers (UTIs).
- Use of the more advanced Core Foundation memory model (rather than using `malloc`).

See "Scrap Manager Versus the Pasteboard Manager" (page 31) for a listing of Scrap Manager functions and their suggested Pasteboard Manager replacements.

Pasteboard Items

Each piece of data placed onto a pasteboard is considered a pasteboard item. Each item has a unique item ID associated with it. This item ID is arbitrarily determined by the application adding to the pasteboard and can be any value that lets the application keep track of the pasteboard data. For example, an item ID could be the memory address of the data, a hash table key, or simply an index value (0, 1,2 and so on.).

The pasteboard can hold multiple items. Applications can place or retrieve as many items as they wish. For example, say a user selection in a browser window contains both text and a GIF image. The Pasteboard Manager lets you copy the text and the image to the pasteboard as separate items. An application pasting multiple items can choose to take only those that is supports (the text, but not the image, for example).

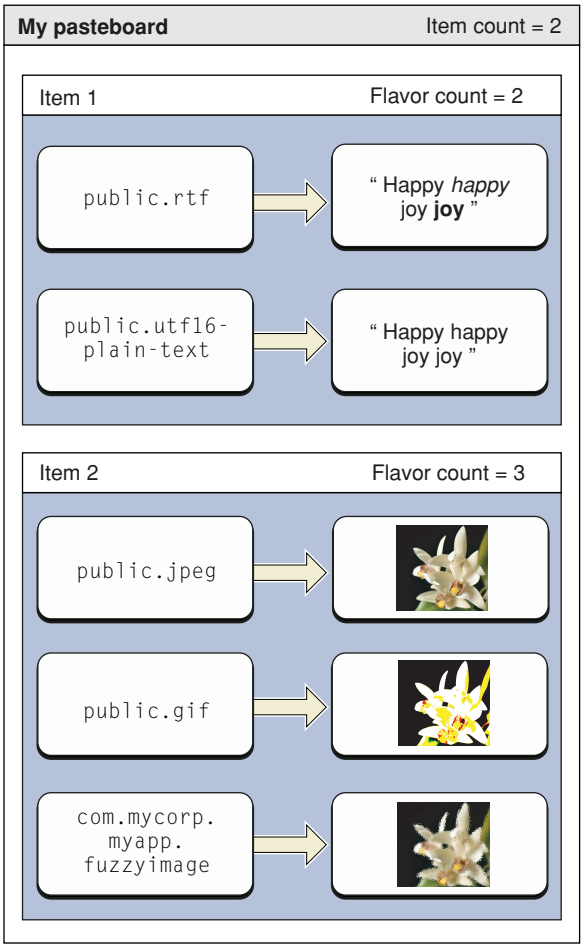
Data Flavors

Each item on the pasteboard can have one or more flavors. A flavor is simply an identifier indicating what type of data is available. For example, a flavor can identify a chunk of data as a JPEG file. Applications retrieving data from the pasteboard use flavors to determine which items they want to support. For example, an image editing program might support pasting of multiple image types (JPEG, TIFF, PICT, and so on) while a command line editor might only allow pasting of text.

The Pasteboard Manager uses uniform type identifiers (UTIs) to specify item flavors. A UTI is simply a Core Foundation string that uniquely identifies a particular data type. For more information, see *Uniform Type Identifiers Overview* in Carbon Interapplication Communication documentation and `UTTypes.h` in the Launch Services framework.

Any item on a pasteboard may be represented by multiple flavors, to make it easier for whatever application wants to retrieve it. For example, an application may want to place both plain text and Unicode versions of a text selection on a pasteboard. Each flavor variant is stored with its own data, as shown in Figure 1-1 (page 11).

Figure 1-1 Pasteboard items and data flavors



Flavors also have flags associated with them that specify additional useful information or properties. Some example flags are as follows:

- Only the application or process that added the flavor can see it. This flag is useful for indicating proprietary information that should only be handled wthin the owning application.
- The flavor is hidden from the public, but may be retrieved if the paste recipient specifically asks for it. A real-world analogy might be a special delicacy that isn't listed on a restaurant menu. Customers "in the know" can request the dish, but the restaurant does not advertise it.
- The data associated with this flavor is promised. See "Promised Data" (page 12).
- The flavor is available through a translation service. That is, the flavor data does not currently exist, but an available flavor can be converted to this data type. See "Handling Translations Using Pasteboards" (page 25) for more information.

Promised Data

Promised data is handled just like any other data added to the pasteboard, except that you don't actually put the data there. You specify their flavors and flags as usual, but instead of adding the actual data, you add a promise to deliver the data at a later time.

For a real-world analogy, say you wanted to use a public bulletin board to give away some furniture. Because it is impractical to pin the individual tables, chairs, or sofas, to the board, you would instead post a note with your phone number advertising free furniture. Interested parties could then contact you and arrange for delivery or pickup of the items.

Similarly, you promise data to the pasteboard when placing the actual data is either impractical or time-consuming. For example, say your application places a JPEG image on the pasteboard. For maximum compatibility, it might be useful to add alternate flavors of the image as well (TIFF, GIF, and so on) to support applications that don't support JPEGs. However, because the alternate data forms do not currently exist, it is more efficient to promise the alternate flavors rather than taking the overhead of generating them on the off chance that some application will want one. If a paste recipient wants the GIF version, your application can generate it after it is requested.

To honor a promise, an application must register a promise keeper callback function. When a paste recipient requests a promised flavor, your callback must then supply the actual data. If you have multiple promised items on the pasteboard, your promise keeper callback must be able to supply the data for any of the indicated promises.

Pasteboard Manager Tasks

This chapter describes how to implement pasteboards in your application

A Day in the Life of a Pasteboard

To get a better understanding of how the Pasteboard Manager works, it's useful to understand the steps inherent in its use. For example, here is the general sequence that underlies a copy and paste action.

The copying application is responsible for placing copied or cut data onto the pasteboard:

1. The user selects some data and invokes the Copy (or Cut) menu item.
2. If the application doesn't already have a reference to the Clipboard pasteboard, it creates one.
3. The application then takes ownership of the pasteboard and clears the current contents.
4. The application assigns item IDs to the selected data.
5. If any data is to be promised, the application must register a promise keeper callback function to supply the promised data.
6. The application adds one or more flavors of each item to the pasteboard, including either the actual flavor data or a promise with each flavor.

The receiving application has a slightly different set of tasks to handle the Paste action:

1. When the application becomes active, it checks to see if the pasteboard has been modified. If so, it obtains a listing of the flavors on the pasteboard. If there are any flavors the application supports, it can enable its Paste menu item.
2. The user invokes the Paste menu item.
3. The application requests the item (or items) on the pasteboard in the flavors that it supports.
4. If the pasted data is to be stored as a file, the receiving application needs to set a paste location before requesting any flavor data. In any other case, the receiving application doesn't need to worry about whether the paste data was promised or not.

If the copying application's promise keeper is called, the callback must do the following:

- If the data is to be stored as a file, determine the paste location specified by the receiving application.
- Generate or otherwise prepare the promised data for transfer.
- If the promised data is not be stored as a file, add the flavor and data to the pasteboard. Otherwise, transfer the promised data to the specified file location.

Some time later, when the application quits, or when it no longer needs the pasteboard, the application can release the pasteboard reference.

Creating a Pasteboard

To create a pasteboard, you call the function `PasteboardCreate`. The name is a bit misleading, as it may not actually create a pasteboard. This function does one of two things:

- If the pasteboard already exists, it creates a retained reference to that pasteboard, which your application can subsequently use to access it.
- If the pasteboard does not exist, `PasteboardCreate` creates it for you and then creates a reference to it.

In most cases your application does not need to worry about which of these actions occurred, because it obtains a pasteboard reference in either case.

```
OSStatus err = noErr;
PasteboardRef theClipboard;
    err = PasteboardCreate( kPasteboardClipboard, &theClipboard );
```

In this example, the constant `kPasteboardClipboard` indicates that you want to reference the Clipboard pasteboard. Passing `kPasteboardFind` instead specifies the Find pasteboard. You can also specify custom pasteboards in one of two ways:

- You can pass a unique pasteboard name. This name should be a Core Foundation string presented in reverse-DNS format to ensure uniqueness (that is, in the form `com`.*myCorp.myApp.myPasteboard*).
- You can pass the constant `kPasteboardUniqueName`, which lets the Pasteboard Manager creates a uniquely-named pasteboard on your behalf. This option is useful if the pasteboard will have a short lifespan, or will not be widely shared.

As stated previously, pasteboards live in a special global memory space so that they can be shared between applications. Unless you explicitly want a custom pasteboard to exist after your application quits, you should release the pasteboard reference by calling `CFRelease` when you are done using it. The Clipboard and Find pasteboards are always retained by the system and will never disappear.

Adding Data to the Pasteboard

After you have created a pasteboard, you can add data to it. For example, you would add data to the pasteboard when the user selects some data and then chooses the Copy or Cut menu item. You can then handle the copy operation in your `kHICommandCopy` event handler.

Listing 1-1 (page 15) shows how you might add some text to a pasteboard. This example assumes that the selection to be added to the pasteboard is contained within an MLTE text object and that the selection is considered to be a single pasteboard item.

Listing 2-1 Adding data to a pasteboard

```
OSStatus AddDataToPasteboard( PasteboardRef inPasteboard,
                                                TXNObject inTXNObject){
OSStatus            err = noErr;
PasteboardSyncFlags syncFlags;
TXNOffset           start, end;
Handle              dataHandle;
CFDataRef           textData = NULL;

err = PasteboardClear( inPasteboard );                                          // 1
require_noerr( err, CantClearPasteboard );

syncFlags = PasteboardSynchronize( inPasteboard );                              // 2
require_action( !(syncFlags&kPasteboardModified),                               // 3
             PasteboardNotSynchedAfterClear, err = badPasteboardSyncErr );
require_action( (syncFlags&kPasteboardClientIsOwner),
             ClientNotPasteboardOwner, err = notPasteboardOwnerErr );

    TXNGetSelection( inTXNObject, &start, &end );                               // 4
    err = TXNGetDataEncoded( inTXNObject, start, end, &dataHandle,              // 5
                                                    kTXNUnicodeTextData );
    require_noerr( err, CantGetDataFromTextObject );

    textData = CFDataCreate( kCFAllocatorDefault,                               // 6
                                    (UInt8*)*dataHandle, (end-start)*2 );
    DisposeHandle( dataHandle );
    require_action( textData != NULL, CantCreateTextData, err = memFullErr );

    err = PasteboardPutItemFlavor( inPasteboard, (PasteboardItemID)1,           // 7
                        CFSTR("public.utf16-plain-text"),
                        textData, 0 );
    require_noerr( err, CantPutTextData );

CantPutTextData:
CantCreateTextData:
CantGetDataFromTextObject:
CantSetPromiseKeeper:
ClientNotPasteboardOwner:
PasteboardNotSynchedAfterClear:
CantClearPasteboard:

    return err;
}
```

Here is how the code works:

1. `PasteboardClear` clears the current contents of the pasteboard and makes your application the pasteboard owner. You must clear the pasteboard before you can add any data of your own.

2. You normally call `PasteboardSynchronize` when your application becomes active, in order to determine if the pasteboard contents have changed. However, it can also be useful to call this function from a plugin to determine if the host application has ownership of the pasteboard. If so, the plugin can go ahead and add data.

3. This code example and others in this section use Apple-defined error-checking macros. For more information about using these macros, see the header `AssertMacros.h` in `/usr/include`.

4. Call the MLTE function `TXNGetSelection` to obtain the start and end points of the selected text in the text object.

5. Call the MLTE function `TXNGetDataEncoded` to extract the selection as non-Unicode text, which is then placed in `dataHandle`.

6. Pasteboards require data to be store as `CFData` objects, so you need to call `CFDataCreate` to create a `CFData` object containing the text data. After creating the object, you can release the original text data. Note that Unicode encoding requires two bytes per character.

7. Call `PasteboardPutItemFlavor` to place a flavor-variant of the selected text on the pasteboard. Each piece of data must have a flavor associated with it (plain text in this case), and an item ID. In this example, the item ID is simply set as 1, but you can assign any value you wish. If you wanted to add another flavor of the same selected text (say Unicode text), you would specify the same item ID, but a different flavor.

   If you want to add multiple items, you should generate them here and then call `PasteboardPutItemFlavor` to add each flavor of the item to the pasteboard.

Each item you put onto a pasteboard must have a unique item ID. However, an item may be available in several different flavors. For example, you can call `PasteboardItemPutFlavor` several times specifying the same item ID, each time with a different flavor.

Handling Pasteboard Promises

If you want to promise data when adding a flavor, you must pass `kPasteboardPromisedData` for the data parameter when calling `PasteboardPutItemFlavor`, In addition, before calling `PasteboardPutItemFlavor`, you must have set a promise keeper callback function by calling `PasteboardSetPromiseKeeper`.

```
PasteboardSetPromiseKeeper( inPasteboard, myPromiseKeeperCallback,
                        inContextData );
```

The `inContextData` parameter can hold any data that you would like to have passed to your callback function . When a receiving application attempts to obtain promised data from the pasteboard (by calling `PasteboardCopyItemFlavorData`) the Pasteboard Manager calls your promise keeper callback function.

Listing 1-2 (page 16) shows you might implement a promise keeper callback.

Listing 2-2 A simple pasteboard promise keeper callback function

```
OSStatus myPromiseKeeperCallback ( PasteboardRef inPasteboard,                // 1
                PasteboardItemID inItem, CFStringRef inFlavorType,
                void *inContext )
{
    OSStatus    err = noErr;
    CFDataRef   promisedData = NULL;

    // create the promised flavor data here, as a CFData object               // 2
…

    err = PasteboardPutItemFlavor( inPasteboard, inItem, inFlavorType,        // 3
                                promisedData, 0 );
```

```
    CFRelease( promisedData );

    require_noerr( err, CantFulfillPromise );

CantFulfillPromise:
CantCreatePromisedData:

    return err;
}
```

Here is how the code works:

1. When your promise keeper callback is called, it receives the item ID and the flavor type that you specified when you called `PasteboardPutItemFlavor`. It also passes you any context data that you specified when you called `Pasteboard SetPromiseKeeper`. This information should be enough to determine what promised data you need to generate.
2. You need to convert your promised data to a `CFData` object before you can add it to a pasteboard. Typically you do so by calling the Core Foundation function `CFDataCreate`.
3. Call `PasteboardPutItemFlavor` to put the promised data onto the pasteboard.

In most cases, you simply add the promised data to the pasteboard just as you would add any other flavor data. However, in some cases the receiver may have specified a URL (by calling `PasteboardSetPasteLocation` indicating where to place the data. In such cases, your callback needs to call `PasteboardCopyPasteLocation` to determine the drop location. For example, the Finder lets you copy and paste files in the file system. Rather than moving the contents of the files to the clipboard, it is simpler for the receiver to set the desired paste location. The Finder can then copy the files directly rather than going through the pasteboard.

Retrieving Data from the Pasteboard

Prior to doing any actual pasting, your application should call `PasteboardSynchronize` when handling the `kEventAppActivated` event to determine if the pasteboard has been modified. If it has, your handler should iterate through the available item flavors and provide visual feedback (activate the Paste menu item, highlight the drag target, and so on) if any compatible data exists. To improve efficiency, your application can exit the search loop immediately upon finding any compatible flavor; identifying all the pasteable flavors can wait until the actual paste routine. Listing 1-5 (page 22) shows a possible implementation of the search loop as used in a drag tracking handler.

Listing 1-3 (page 17) shows how an application might retrieve data from a pasteboard.

Listing 2-3 Obtaining data from the pasteboard

```
OSStatus GetDataFromPasteboard( PasteboardRef inPasteboard, TXNObject inTXNObject )
{
    OSStatus            err = noErr;
    PasteboardSyncFlags syncFlags;
    ItemCount           itemCount;

    syncFlags = PasteboardSynchronize( inPasteboard );                          // 1
    require_action( syncFlags&kPasteboardModified, PasteboardOutOfSync,
```

```
                err = badPasteboardSyncErr );

    err = PasteboardGetItemCount( inPasteboard, &itemCount );                          // 2
    require_noerr( err, CantGetPasteboardItemCount );

    for( UInt32 itemIndex = 1; itemIndex <= itemCount; itemIndex++ )
    {
        PasteboardItemID    itemID;
        CFArrayRef          flavorTypeArray;
        CFIndex             flavorCount;

        err = PasteboardGetItemIdentifier( inPasteboard, itemIndex, &itemID );         // 3
        require_noerr( err, CantGetPasteboardItemIdentifier );

        err = PasteboardCopyItemFlavors( inPasteboard, itemID, &flavorTypeArray );     // 4
        require_noerr( err, CantCopyPasteboardItemFlavors );

        flavorCount = CFArrayGetCount( flavorTypeArray );                              // 5

        for( CFIndex flavorIndex = 0; flavorIndex < flavorCount; flavorIndex++ )
        {
            CFStringRef             flavorType;
            CFDataRef               flavorData;
            CFIndex                 flavorDataSize;
            char                    flavorText[256];


            flavorType = (CFStringRef)CFArrayGetValueAtIndex( flavorTypeArray,         // 6
                                                              flavorIndex );

            if (UTTypeConformsTo(flavorType, CFSTR("public.utf16-plain-text")))        // 7
            {

                err = PasteboardCopyItemFlavorData( inPasteboard, itemID,              // 8
                                                    flavorType, &flavorData );
                require_noerr( err, CantCopyFlavorData );

                flavorDataSize = CFDataGetLength( flavorData );


                flavorDataSize = (flavorDataSize<254) ? flavorDataSize : 254;
                for( short dataIndex = 0; dataIndex <= flavorDataSize; dataIndex++ )   // 9
                    {
                    char byte = *(CFDataGetBytePtr( flavorData ) + dataIndex);
                    flavorText[dataIndex] = byte;
                    }
                flavorText[flavorDataSize] = '\0';
                flavorText[flavorDataSize+1] = '\n';
                TXNSetData( inTXNObject, kTXNTextData, flavorText,                     // 10
                            flavorDataSize+2, kTXNEndOffset, TXNEndOffset );

                CFRelease (flavorData);
            }

CantCopyFlavorData:
            ;
        }
```

```
    CFRelease (flavorTypeArray);

CantCopyPasteboardItemFlavors:
CantGetPasteboardItemIdentifier:
        ;
    }

CantGetPasteboardItemCount:
PasteboardOutOfSync:

    return err;
}
```

Here is how the code works:

1. In most cases, you would call `PasteboardSynchronize` from your `kEventAppActivated` handler to synchronize your pasteboard reference with the actual global pasteboard. However, you can call it here if your application does not do any flavor checking beforehand.
2. Call `PasteboardGetItemCount` to determine the number of items in the pasteboard.
3. For each item index, call `PasteboardGetItemIdentifer` to obtain the item ID.
4. Once you have the item ID, call `PasteboardCopyItemFlavors` to obtain the array of flavors corresponding to that item.
5. The Core Foundation function `CFArrayGetCount` determines the number of flavors in the array.
6. Next, iterate through the array of flavors, calling the Core Foundation function `CFArrayGetItemAtIndex` to obtain the flavor strings.
7. Use the uniform type identifier function `UTTypeConformsTo` to determine if the flavor is compatible with the plain Unicode text flavor type. UTIs are arranged in a conformance hierarchy (similar to a class hierarchy) which determines compatibility between flavors.
8. If the flavor is compatible, retrieve the flavor data from the pasteboard by calling `PasteboardCopyItemFlavorData`.
9. Iterate through the bytes of the data and copy them into the `flavorText` string. Note that because the data is Unicode, each character takes two bytes.
10. Call `TXNSetData` to add the pasted data to the end of the MLTE text object.

The way this example is written, it will add every flavor that conforms to the plain Unicode text UTI to the MLTE object. In most cases, your application will choose to obtain only one flavor of each pasteboard item.

Using the Pasteboard In Drag-And-Drop Operations

In Mac OS X v10.3 and later, the Drag Manager can use the Pasteboard Manager to transfer data in drag-and-drop operations. The mechanism is very similar to using a pasteboard to facilitate a cut-and-paste operation. When a drag occurs, the client application creates a pasteboard and places the dragged data onto it. To accept a drop, the receiving application obtains the data from the pasteboard.

Before implementing drag-and-drop in your application, be sure you understand and follow the drag-and-drop guidelines specified in the *Apple Human Interface Guidelines*.

Initiating a Drag

When the user begins a drag, your application receives a `kEventControlTrack` event. Your application's tracking handler should then place the dragged data onto a pasteboard.

The drag initiation handler in Listing 1-4 (page 20) uses the Pasteboard Manager to hold the MLTE text selection being dragged.

Listing 2-4 A drag initiation handler that uses the Pasteboard Manager

```
OSStatus HandleDragInitiation( EventRef inEvent, TXNObject inTXNObject )
{
    OSStatus    err = eventNotHandledErr;
    EventRecord convertedEvent;
    UInt32      keyModifiers;
    HIPoint     eventPoint;
    TXNOffset   start, end, offset;

    require_noerr( GetEventParameter( inEvent, kEventParamMouseLocation,                // 1
     typeHIPoint, NULL, sizeof( eventPoint ), NULL, &eventPoint ), CantGetHIPoint );

    require_noerr( TXNHIPointToOffset( inTXNObject, &eventPoint, &offset ),             // 2
                                                    CantGetTXNOffsetForPoint );

    TXNGetSelection( inTXNObject, &start, &end );                                       // 3

    convertedEvent.what = mouseDown;                                                    // 4
    GetGlobalMouse (&(convertedEvent.where));
    GetEventParameter (inEvent, kEventParamKeyModifiers, typeUInt32, NULL,
                       sizeof (keyModifiers), NULL, &keyModifiers);
    convertedEvent.modifiers = keyModifiers;

    if( start <= offset && offset <= end && WaitMouseMoved( convertedEvent.where ) )    // 5
    {
        PasteboardRef   pasteboard;
        DragRef         drag;
        RgnHandle       theRegion, insetRegion;

        err = PasteboardCreate( kPasteboardUniqueName, &pasteboard );                   // 6
        require_noerr( err, CantCreatePasteboardForDrag );

        err = AddDataToPasteboard( pasteboard, inTXNObject);                            // 7
        require_noerr( err, CantAddDataToTheDragPasteboard );

        err = NewDragWithPasteboard( pasteboard, &drag );                               // 8
        require_noerr( err, CantCreateDrag );

        theRegion = NewRgn();                                                           // 9
        insetRegion = NewRgn();
       SetRectRgn( theRegion, convertedEvent.where.h - 10, convertedEvent.where.v - 5,
                       convertedEvent.where.h + 10, convertedEvent.where.v + 5 );
        CopyRgn( theRegion, insetRegion );
```

```
        InsetRgn( insetRegion, 1, 1 );
        DiffRgn( theRegion, insetRegion, theRegion );
        DisposeRgn( insetRegion );

        err = TrackDrag( drag, &convertedEvent, theRegion );                    // 10
        require_noerr( err, CantTrackDrag );

CantTrackDrag:

        DisposeRgn( theRegion );
        check_noerr( DisposeDrag( drag ) );

CantCreateDrag:
CantAddDataToTheDragPasteboard:

        CFRelease( pasteboard );                                                // 11
    }

CantCreatePasteboardForDrag:
CantGetTXNOffsetForPoint:
CantGetHIPoint:

    return err;
}
```

Here is how the code works:

1. Call `GetEventParameter` to obtain the mouse position from the `kEventControlTrack` event.
2. The MLTE function `TXNGetOffsetFromPoint` determines the position of the mouse as a character offset within the text object.
3. Use the MLTE function `TXNGetSelection` to obtain the beginning and end offsets for the text selection within the text object.
4. The Drag Manager `TrackDrag` function still assumes that event information is stored in old Event Manager event records. A simple way around this is to manually populate an event record with the relevant data. This example fills out the event record with the event type, `mouseDown`, the mouse position, and the keyboard modifiers.
5. If the mouse offset is within the text selection, and the user has begun to drag the mouse (`WaitMouseMoved` returns `True`), then you can initiate the actual drag.
6. Call `PasteboardCreate` to create a unique pasteboard. This pasteboard needs to exist only for the length of the drag-and-drop operation.
7. To add the selection data to the pasteboard, call the `AddDataToPasteboard` function described in Listing 1-1 (page 15), passing the pasteboard and the text object containing the text to be dragged.
8. To initiate the actual drag, create a drag object using the Drag Manager function `NewDragWithPasteboard`.
9. Set up a region to be the visual feedback for the drag. This example sets up a small rectangular region to be dragged. However, for most applications, the region should contain a translucent image or other similar rendering of the data being dragged.
10. Call the Drag Manager function `TrackDrag` to begin tracking the actual drag.

11. After you have initiated the actual drag, you no longer need to hold onto the pasteboard reference (the Drag Manager retains it), so you can release it by calling `CFRelease`.

Tracking a Drag

During drag tracking, if the drag enters a droppable region, the receiving application must determine whether it can accept the drag and, if so, highlight the region. You do so using a drag tracking callback function.

You install your drag tracking callback on a window using the Drag Manager function `InstallTrackingHandler`:

```
OSErr InstallTrackingHandler(
  DragTrackingHandlerUPP   trackingHandler,
  WindowRef                theWindow,
  void *                   handlerRefCon
);
```

In this example, you would pass the text view to receive the drag in the reference constant parameter.

Your drag tracking handler should visually indicate whether the data being dragged into the window is something it can accept. Listing 1-5 (page 22) gives an example of how you might do this.

Listing 2-5 Tracking a drag in a window

```
OSErr DragTrackingHandler( DragTrackingMessage inMessage, WindowRef inWindow,
                                            void *inUserData, DragRef inDrag )
{
    OSStatus err = noErr;
    Boolean tastyFlavor = false;                                                    // 1

    switch( inMessage )
    {
        case kDragTrackingEnterWindow:                                              // 2
        {
            ItemCount itemCount;
            DragAttributes attributes;
            PasteboardRef pasteboard;

            GetDragAttributes( inDrag, &attributes );                               // 3

            err = GetDragPasteboard( inDrag, &pasteboard );                         // 4
            require_noerr( err, CantGetDragPasteboard );

            err = PasteboardGetItemCount (pasteboard, &itemCount);                  // 5

            for (UInt32 itemIndex = 1;
                                (itemIndex <= itemCount) && !tastyFlavor; itemIndex++)
            {
                PasteboardItemID itemID;
                CFArrayRef  flavorTypeArray;
                CFIndex     flavorCount;

                err = PasteboardGetItemIdentifier (pasteboard, itemIndex, &itemID);
                require_noerr (err, CantGetPBItemIdentifier);
```

```
            err = PasteboardCopyItemFlavors (pasteboard, itemID, &flavorTypeArray);
            require_noerr (err, CantGetPBItemFlavors);

            flavorCount = CFArrayGetCount(flavorTypeArray);

            for (CFIndex FlavorIndex = 0;
                    (FlavorIndex < flavorCount) && !tastyFlavor; FlavorIndex++)
            {
                CFStringRef flavorType;

                flavorType = (CFStringRef)
                        CFArrayGetValueAtIndex (flavorTypeArray, FlavorIndex);

                if (UTTypeConformsTo (flavorType,
                                      CFSTR("public.plain-text")))
                     tastyFlavor = true;
            }

            CFRelease (flavorTypeArray);

            CantGetPBItemIdentifier:
            CantGetPBItemFlavors:
                ;
        }
        if( (attributes & kDragHasLeftSenderWindow) && tastyFlavor )              // 6
        {
            HIViewRef   textView = (HIViewRef)inUserData;
            HIRect      textFrame;
            RgnHandle   hiliteRgn = NewRgn();

            HIViewGetFrame( textView, &textFrame );                               // 7

            HIShapeRef textShape = HIShapeCreateWithRect( &textFrame );           // 8
            HIShapeGetAsQDRgn( textShape, hiliteRgn );                            // 9
            CFRelease( textShape );

            ShowDragHilite( inDrag, hiliteRgn, true );                            // 10

            DisposeRgn( hiliteRgn );
        }

        CantGetDragPasteboard:
        ;
    }
    break;

    case kDragTrackingLeaveWindow:
    {
        HideDragHilite( inDrag );                                                 // 11
    }
    break;
}

return err;
}
```

Here is how the code works:

1. Define the `tastyFlavor` flag. Initially set to `false`, the tracking handler sets this flag if it finds a data flavor that the application can accept.

2. If the user drags data into the window, the tracking handler receives the `kDragTrackingEnterWindow` message.

3. Obtain the drag attributes by calling the Drag Manager function `GetDragAttributes` These attributes are needed later to determine whether or not to highight the accepting window.

4. Call the Drag Manager function `GetDragPasteboard` to obtain the pasteboard associated with the drag (created in the drag initiation handler (Listing 1-4 (page 20))).

5. Determine if the drag contains a flavor the application can accept. This code is essentially a stripped-down version of the flavor-checking code in Listing 1-3 (page 17). This example simply determines if a compatible flavor exists, and if so, sets the `tastyFlavor` flag and drops out of the loops.

   Note however, that to conform the to the Apple Human Interface Guidelines for receiving drags, your application must be able to accept a flavor from each dragged item in order to highlight the receiving view. See Destination Feedback in the *Apple Human Interface Guidelines* for more details.

6. Determine whether to highlight the receiving view or not. To be eligible for highlighting, the drag must have cleared the sending application's window (as dictated by the Apple Human Interface Guidelines) and an acceptable flavor must exist (`tastyFlavor== true`).

7. Obtain the frame coordinates of the receiving view (a text view in this example) by calling the HIView function `HIViewGetFrame`.

8. Call the HIShape function `HIShapeCreateWithRect` to create an HIShape with the same coordinates as the text view.

9. Call the HIShape function `HIShapeGetAsQDRgn` to convert the HIShape into an classic QuickDraw region, as that is what the drag highlighting function expects. After creating this region, release the no longer needed HIShape by calling the Core Foundation function `CFRelease`.

10. Call the Drag Manager function `ShowDragHilite` to highlight the text view region. After passing the region to `ShowDragHilite`, you are free to dispose it.

11. The other interesting drag message this callback handles is `kDragTrackingLeaveWindow`, sent when the user moves the dragged data out of the receiving window. In this case, the callback simply hides the drag highlight by calling the Drag Manager function `HideDragHilite`.

The overhead of obtaining pasteboard items and flavors is relatively low, so it is generally not a problem to repeat the flavor search each time the drag enters the view as well as when you actually obtain the pasteboard data.

Receiving a Drag

When the user completes a drag-and-drop operation, the receiving application retrieves the data from the pasteboard in the standard drag receive callback function. You install your callback on a window using the Drag Manager function `InstallReceiveHandler`:

```
OSErr InstallReceiveHandler (
   DragReceiveHandlerUPP receiveHandler,
```

```
    WindowRef theWindow,
    void * handlerRefCon
);
```

In this example, you would pass the text object associated with the view receiving the drag as the `handlerRefCon` parameter, which is then passed to the drag receive handler in Listing 1-6 (page 25).

Listing 2-6 Receiving a drag using the pasteboard

```
OSErr DragReceiveHandler( WindowRef inWindow, void *inUserData,
                          DragRef inDrag )
{
    OSStatus        err = noErr;
    PasteboardRef   pasteboard;
    TXNObject       txnObject = (TXNObject)inUserData;

    err = GetDragPasteboard( inDrag, &pasteboard );                    // 1
    require_noerr( err, CantGetDragPasteboard );

    err = GetDataFromPasteboard( pasteboard, txnObject );              // 2

    CantGetDragPasteboard:

    HideDragHilite( inDrag );                                          // 3
    return err;
}
```

Here is how the code works:

1. Call the Drag Manager function `GetDragPasteboard` to obtain the pasteboard associated with the drag.
2. Get the data from the pasteboard, just as if you were obtaining it for a paste operation. This example simply calls the `GetDataFromPasteboard` function in Listing 1-3 (page 17) to put the pasteboard data into a text object.
3. After the drop operation is over, call the Drag Manager function `HideDragHilite` to remove the window highlighting added by the drag tracking callback.

Handling Translations Using Pasteboards

The Pasteboard Manager supports data translators that present additional flavors available on a pasteboard. For example, a text-editing application may make two data flavors available when it places selected text onto the Clipboard pasteboard, plain and rich (RTF). A registered filter service can also make available another flavor, say `com.apple.mytranslatorapp.uppercasetext` that is translated from the plain text flavor. That is, when the receiving application obtains the array of data flavors to determine if it can handle them, the upper-case option is included as a promised flavor. These services are automatically available on any pasteboard you create or reference using the Pasteboard Manager.

If the receiving application chooses to obtain the upper-case text, the plain text is placed onto a new pasteboard and the filter service is invoked. The service obtains the raw data from the pasteboard, translates it accordingly, and then puts the translated data back on the pasteboard. The translated text is then placed back onto the Clipboard pasteboard where the the receiving application can obtain it.

If your application simply uses pasteboards for data sharing, any translation of pasteboard data occurs transparently. However, if you want to make a translation filter available, you need to write and register a filter service.

For example, say you want to implement a service that handles data translation (also called filtering) from standard text to all upper-case lettering. You must register this translation service by defining an NSFilter service in your application's `Info.plist` file, such as follows:

```
<key>NSServices</key>
    <array>
        <dict>
            <key>NSFilter</key>
            <string>UpperCaseTranslation</string>
            <key>NSReturnTypes</key>
            <array>
                <string>com.apple.mytranslatorapp.uppercasetext</string>
            </array>
            <key>NSSendTypes</key>
            <array>
                <string>public.plain-text</string>
            </array>
            <key>NSSupportsDataTranslation</key>
            <string></string>
        </dict>
    </array>
```

This defines an NSFilter that can take data of type `public.plain-text` and translate it to data of type `com.apple.mytranslatorapp.uppercasetext`. Of course, the returned data can be any standard type as well.

Whenever data is placed onto a pasteboard, the Services Manager scans the flavors and checks them against its list of registered translators. If a translator exists for a given flavor, it can promise the translated flavor on the pasteboard. However, translators are not transitive; if translators exist to convert type A to B and type B to C, this does not mean that a translator from A to C exists. If the receiving application requests a translated flavor, the service that offers that translation receives a `kEventServicePerform` event.

The scope of the translation is entirely up to the filter creator. For example, you could create a filter to change plain text to Unicode text, turn encapsulated PostScript(EPS) images into GIFs, or even translate English into French. The only restriction is that you must be able to present the translation option as a unique UTI.

Listing 1-7 (page 26) shows how you might implement a filter service that offers text translation to either all upper-case or all lower-case lettering.

Listing 2-7 A filter service handler

```
OSStatus HandlePerformService( EventRef inEvent )                    // 1
{
    OSStatus            err = noErr;
    CFStringRef         serviceName, returnType;
    PasteboardRef       pasteboard;
    PasteboardItemID    item;
```

```
    CFDataRef           sourceData;
    CFIndex             sourceSize;
    CFMutableDataRef    returnData = NULL;
    const UInt8*        sourceBytes;
    UInt8*              returnBytes;

    err = GetEventParameter( inEvent, kEventParamServiceMessageName,          // 2
         typeCFStringRef, NULL, sizeof( CFStringRef ), NULL, &serviceName );
    require_noerr( err, CantGetServiceName );

    err = GetEventParameter( inEvent, kEventParamPasteboardRef,               // 3
         typePasteboardRef, NULL, sizeof( PasteboardRef ), NULL,
         &pasteboard );
    require_noerr( err, CantGetPasteboardRef );

    err = PasteboardGetItemIdentifier( pasteboard, 1, &item );                // 4
    require_noerr( err, CantGetItemIdentifier );

    err = PasteboardCopyItemFlavorData( pasteboard, item,                     // 5
                CFSTR("public.plain-text"), &sourceData );
    require_noerr( err, CantGetSourceData );

    sourceSize = CFDataGetLength( sourceData );                               // 6

    returnData = CFDataCreateMutable( kCFAllocatorDefault, sourceSize );      // 7
    require_action( returnData != NULL, CantCreateReturnData,
                    err = memFullErr );

    sourceBytes = CFDataGetBytePtr( sourceData );
    returnBytes = CFDataGetMutableBytePtr( returnData );
    CFDataSetLength( returnData, sourceSize );

    if( CFStringCompare( serviceName, CFSTR("UpperCaseTranslation"), 0 )
            == kCFCompareEqualTo )
    {
        returnType = CFSTR("com.apple.pasteboardpeeker.uppercasetext");       // 8

        for( CFIndex i=0; i<sourceSize; i++ )                                 // 9
            returnBytes[i] = (UInt8)toupper( sourceBytes[i] );
    }
    else
    {
        returnType = CFSTR("com.apple.pasteboardpeeker.lowercasetext");       // 10

        for( CFIndex i=0; i<sourceSize; i++ )
            returnBytes[i] = (UInt8)tolower( sourceBytes[i] );
    }

    err = PasteboardClear( pasteboard );                                      // 11
    require_noerr( err, CantClearPasteboard );

    // add the translated data
    err = PasteboardPutItemFlavor( pasteboard, item, returnType,              // 12
            returnData, 0 );
    require_noerr( err, CantAddTranslatedData );

CantAddTranslatedData:
```

```
CantClearPasteboard:

    CFRelease( returnData );

CantCreateReturnData:

    CFRelease( sourceData );

CantGetSourceData:
CantGetItemIdentifier:
CantGetPasteboardRef:
CantGetServiceName:

    return err;
}
```

Here is how the code works:

1. The `HandlePerformService` function receives an event reference from the `kEventServicePerform` event handler that calls it.
2. Call `GetEventParameter` to obtain the service message that generated the perform event.
3. For service events, a pasteboard storing the data to be filtered is stored in the event. Call `GetEventParameter` specifying the `kEventParamPasteboardRef` parameter to obtain it. This pasteboard was allocated for you, so you do not need to release it when you are done with it.
4. This example assumes that the data to be filtered is the first item in the pasteboard.
5. This example also assumes that the data in the item is of type `public.plain-text`.
6. Call the Core Foundation function `CFGetDataLength` to determine the size of the data. As the text contains single-byte characters, this is also the length of the text string.
7. Call the Core Foundation function `CFDataCreateMutable` to allocate memory to hold the translated string data.
8. If the service request is to translate to upper-case, set the return type of the translated text to a unique UTI reflecting this. You must do this, because Translation Services looks for the translated type on the pasteboard (not the original) when retrieving the data.
9. Use the `toupper` operator to convert the source text to all upper-case lettering.
10. Handle the lower-case option in a similar manner as the upper-case option.
11. Clear the pasteboard in preparation for adding the translated text.
12. Call `PasteboardPutItemFlavor` to add the translated text to the pasteboard. After your service handler completes, the Services Manager puts the translated text back onto the original pasteboard so that the receiving application can retrieve it.

For more information about building and installing services, see *Setting Up Your Carbon Application to Use the Services Menu*.

The Pasteboard Peeker

For additional examples of using pasteboards, you can examine the *PasteboardPeeker* sample code available in the ADC Reference Library.

The Pasteboard Peeker application is also useful for determining what data and types are currently on the Clipboard pasteboard.

Scrap Manager Versus the Pasteboard Manager

If your application still uses the Scrap Manager written for classic Mac OS, Apple encourages you to update your code to use the Pasteboard Manager instead.

The Pasteboard Manager is available in Mac OS X v10.3 and later.

Table A-1 (page 31) lists the Carbon Scrap Manager functions and their Pasteboard Manager equivalents.

Table A-1 Scrap Manager functions and their Pasteboard Manager replacements

Scrap Manager function	Pasteboard Manager equivalent	Comments
`GetScrapByName`	PasteboardCreate	
`GetCurrentScrap`	`PasteboardCreate`	Specify the `kPasteboardClipboard` constant in `PasteboardCreate` to indicate you want the Clipboard.
`GetScrapFlavorFlags`	`PasteboardGetItem-FlavorFlags`	
`GetScrapFlavorSize`	No equivalent	Data is now handed to you as a CFData object, so you no longer need to allocate memory when receiving pasteboard data.
`GetScrapFlavorData`	`PasteboardCopyItem-FlavorData`	
`ClearCurrentScrap`	`PasteboardClear`	Specifically, clear the Clipboard pasteboard.
`ClearScrap`	`PasteboardClear`	
`PutScrapFlavor`	`PasteboardPutItemFlavor`	
`GetScrapFlavorCount`	`PasteboardCopyItemFlavors`	You obtain the flavors as a CFArray, and you can call `CFArrayGetCount` to obtain the number of flavors.
`GetScrapFlavorInfoList`	`PasteboardCopyItemFlavors`	
`CallInScrapPromises`	`PasteboardResolvePromises`	
`SetScrapPromiseKeeper`	`PasteboardSetPromiseKeeper`	
Promise keeper UPP functions	No equivalent	UPPs not required for Mac OS X.

Scrap Manager function	Pasteboard Manager equivalent	Comments
Promise keeper callback function	Promise keeper callback function	The parameter list is different for Pasteboard Manager callbacks.

Document Revision History

This table describes the changes to *Pasteboard Manager Programming Guide*.

Date	Notes
2005-07-07	Fixed typos. Added links to "Uniform Type Identifiers Overview."
2004-12-02	New document that describes how to create and manage pasteboards in Carbon applications.

Document Revision History