Pasteboard Programming Topics for Cocoa

(Legacy)

Developer

Contents

Introduction to Pasteboards Programming Topics

Important: This document may not represent best practices for current development. Links to downloads and other resources may no longer be valid.

You typically use pasteboards in copy and paste operations, although pasteboards also provide the basis of system services (see System Services). `NSPasteboard` objects transfer data to and from the pasteboard server. The server is shared by all running applications. It contains data that the user has cut or copied, as well as other data that one application wants to transfer to another. `NSPasteboard` objects are an application's sole interface to the server and to all pasteboard operations.

You should read this document to learn how to implement copy and paste in your application, and to learn about the different types of pasteboard

Note: This document describes how to use pasteboards on Mac OS X v10.5 and earlier. To understand how to use pasteboards on Mac OS X v10.6 and later, read *Pasteboard Programming Guide for Mac OS X v10.6 and Later*.

Organization of This Document

This document contains the following articles:

- "Pasteboard Fundamentals" (page 6) explains how pasteboards work.
- "Implementing Copy and Paste" (page 9) explains the basics of implementing copy and paste in your application.
- "Named Pasteboards" (page 15) discusses the ability to name pasteboards to reflect their function and the data they contain.
- "Data Types" (page 16) discusses the variety of data that can be placed on pasteboards.
- "Reading and Writing Font Data" (page 18) describes how to work with fonts.
- "Reading and Writing RTFD Data" (page 20) describes how to work with RTFD data.
- "Filter Services" (page 21) discusses the ability of pasteboards to convert data from one type to another using filter services.

- "Providing a Filter Service" (page 22) describes how to create a filter service.

Pasteboard Fundamentals

On Mac OS X, copy and paste operations are supported by a pasteboard server process. In Cocoa, you access the pasteboard server through an `NSPasteboard` object. This article describes how the pasteboard process works.

The basic operations you want to perform when implementing copy and paste are (a) to write data to a pasteboard and (b) to read data from a pasteboard. These operations are conceptually very simple, but mask a number of important details. In practical terms for you as an application developer, the main underlying complicating issue is that there may be a number of ways to represent data—this leads in turn to considerations of efficiency. From the systems perspective, there are additional issues to consider.

The Pasteboard Server

Whether the data is transferred between objects in the same application or two different applications, in a Cocoa application the interface is the same—an `NSPasteboard` object accesses a shared repository where writers and readers meet to exchange data. The writer, referred to as the pasteboard owner, deposits data on a pasteboard instance and moves on. The reader then accesses the pasteboard asynchronously, at some unspecified point in the future. By that time, the writer object may not even exist anymore. For example, a user may have closed the source document or quit the application.

Consequently, when moving data between two different applications, and therefore two different address spaces, a third memory space gets involved so the data persists even in the absence of the source. `NSPasteboard` provides access to a third address space—a pasteboard server process (`pbs`)—that is always running in the background. The pasteboard server maintains an arbitrary number of individual pasteboards to distinguish among several concurrent data transfers.

Named Pasteboards

There are several standard pasteboards provided for well-defined operations system-wide:

- `NSGeneralPboard`—for cut, copy, and paste
- `NSRulerPboard`—for copy and paste of rulers
- `NSFontPboard`—for cut, copy, and paste of `NSFont` objects

- `NSFindPboard`—application-specific find panels can share a sought after text value
- `NSDragPboard`—for graphical drag and drop operations

These are described in more detail in "Named Pasteboards" (page 15). Typically you use one of the system-defined pasteboards, but if necessary you can create your own pasteboard for exchanges that fall outside the predefined set using `pasteboardWithName:` Lastly, if you invoke `pasteboardWithUniqueName`, the pasteboard server will provide you with a uniquely-named pasteboard.

Pasteboard Data Types

The pasteboard owner declares the data types it can write. Pasteboard data generally refer to an object instance whether a string, an arbitrarily complex object graph such as a dictionary of arrays, an instance of `NSData`, or an object wrapper for an arbitrary block of data. You can name your own pasteboard types for special-purpose data types.

Any object written to an `NSPasteboard` must conform to the `NSCoding` protocol—it must be able to archive and unarchive itself. Generally, pasteboard data consists of a value class object or a collection containing such objects.

Like the standard named pasteboards used for common operations, there are several commonly used system-defined data types, including:

- `NSStringPboardType`
- `NSTabularTextPboardType`
- `NSFilenamesPboardType`
- `NSTIFFPboardType`
- `NSFontPboardType`
- `NSRulerPboardType`
- `NSColorPboardType`

These and others are described in detail in "Data Types" (page 16).

Pasteboards Hold Multiple Representations

Pasteboard operations are often carried out between two different applications. For example, an editor, capable of handling rich text format, may allow a user to select a region of text and copy it to the general pasteboard. Another application provides a simple `NSTextView` instance configured to provide ASCII text. It allow the user

to paste from the general pasteboard. Neither application has knowledge about the other and the kinds of data each can handle. It is impossible for the owner to determine which of several applications might show up as the next reader of pasteboard data.

To maximize the potential for sharing, a pasteboard can hold multiple representations of the same data, each identified by a different pasteboard type string. Pasteboard owners should provide as many different representations as possible. In the previous example, the rich text editor might provide RTFD, RTF, and `NSString` representations of the copied data. A reader, on the other hand, must find the data type that best suits its capabilities. Generally, this means selecting the richest type available.

Change Count

The change count is a computer-wide variable that increments every time the contents of the pasteboard changes (a new owner is declared). An independent change count is maintained for each named pasteboard. By examining the change count, an application can determine whether the current data in the pasteboard is the same as the data it last received.

The `changeCount`, `addTypes:owner:`, and `declareTypes:owner:` methods return the change count. A `types` or `availableTypeFromArray:` message should be sent by the pasteboard before reading data so the change count is valid.

Errors

Except where errors are specifically mentioned in the `NSPasteboard` method descriptions, any communications error with the pasteboard server raises an `NSPasteboardCommunicationException`.

Implementing Copy and Paste

This article describes how you can implement copy and paste in your application.

To implement a copy operation you must first tell an `NSPasteboard` object what types of data you want to write, and then you typically write the data, once for each data type. To implement a paste operation, you typically first give the `NSPasteboard` object a list of data types your application can deal with (in your preferred order) and receive back from the pasteboard the identifier for the most preferred type that is actually available. You can then read the data for that type from the pasteboard.

Reading and writing most data types is straightforward, but fonts and RTFD data in particular have some peculiarities—see "Reading and Writing RTFD Data" (page 20) and "Reading and Writing Font Data" (page 18).

Implementing Copy

The first step in implementing a copy method is to decide what representations of your data you want to support. If your application has a custom data type that you want the user to be able to copy and paste within your application, then you need to write a representation of that data to the pasteboard. It may be, though, that you also want to allow the user to paste information from your application into other applications, in which case you need to write your data in a standard representation (such as a string) that other applications can deal with.

Consider an application that allows a user to track expenses. You may have a custom Expense class to represent expense items. In a copy operation, you need to write the currently selected Expense object to the pasteboard. If you want the user to be able to paste the information into another application, such as TextEdit or Mail, then you should also write a textual representation of the data to the pasteboard. If you want to support a custom data type, you must define a name for the type as it will appear on the pasteboard, for example:

```
NSString *ExpensePBoardType = @"ExpensePBoardType";
```

Typically this must be a global value, visible to any objects within your application that will copy this data type to or retrieve it from the pasteboard. Often you assign the variable in an implementation file and declare it as an external variable in a header file that you import from other implementation files:

```
extern NSString *ExpensePBoardType;
```

The first step in copying is to tell the pasteboard (for standard copy and paste operations, this is typically the general pasteboard) what representations you will write (using the method, `declareTypes:owner:`):

```
NSPasteboard *pb = [NSPasteboard generalPasteboard];
NSArray *types = [NSArray arrayWithObjects:
    ExpensePBoardType, NSStringPboardType, NSRTFPboardType, nil];
[pb declareTypes:types owner:self];
```

The owner is typically `self`, and is used if you support lazy initialization (see "Lazy Writing" (page 12)).

You then write the representations to the pasteboard, one at a time. Note that pasteboards support only a limited range of data types, so for custom representations you often need to transform the data into one of the types supported (for example, by archiving an object). In the following example, the Expense class implements custom methods `stringRepresentation` and `rtfRepresentation` to generate a string and RTF representation of the expense respectively.

```
// archive the given expense, and add it to the pasteboard as an Expense
[pb setData:[NSArchiver archivedDataWithRootObject:expense]
    forType:ExpensePBoardType];

// add the string representation
[pb setString:[expense stringRepresentation] forType:NSStringPboardType];

// add the RTF representation
NSAttributedString *rtfDescription = [expense rtfRepresentation];
NSData *rtfData = [rtfDescription
                    RTFFromRange:(NSMakeRange(0, [rtfDescription length]))
                    documentAttributes:nil];
[pb setData:rtfData forType:NSRTFPboardType];
```


Implementing Paste

To implement paste, you first need to determine what data representations are present on the pasteboard, and in particular find the best one available for the situation. You can do this in a single method call, `availableTypeFromArray:`. You pass in an array of types you can support, ordered by preference, and get back the identifier for the best match (assuming there is one, otherwise `nil`). For example, if an Expense class provides a method to parse a string to extract attributes for a new Expense object, you might support both your custom type and the string type:

```
NSArray *pasteTypes = [NSArray arrayWithObjects:
    ExpensePBoardType, NSStringPboardType, nil];
NSString *bestType = [pb availableTypeFromArray:pasteTypes];
if (bestType != nil) {
    // pasteboard has data we can deal with
    // ...
```

Often, though, you support paste operations only for your custom types. In some cases, you might also factor out the code that determines whether the pasteboard contains a supported type:

```
- (BOOL)pasteboardHasExpense {
    // has the pasteboard got an expense?
    NSPasteboard *pb = [NSPasteboard generalPasteboard];
    NSArray *types = [NSArray arrayWithObject:ExpensePBoardType];
    NSString *bestType = [pb availableTypeFromArray:types];
    return (bestType != nil);
}
```

This is useful if you want to support user interface validation, so that for example the Paste menu item is enabled only if the pasteboard contains a data representation you support:

```
- (BOOL)validateUserInterfaceItem:(id <NSValidatedUserInterfaceItem>)item
{
    if ([item action] == @selector(paste:)) {
        return [self pasteboardHasExpense];
    }
    else {
        // ...
```


For more about menu validation, see Enabling Menu Items and *User Interface Validation*.

Assuming that the pasteboard does contain a representation that you support, then you retrieve the corresponding data with `dataForType:`.

```
NSData *data = [pb dataForType:ExpensePBoardType];
Expense *expense = [NSUnarchiver unarchiveObjectWithData:data];
```

Implementing Cut

A cut operation is simply a copy operation followed by a delete operation. In an Expenses application, the `cut:` method might be implemented as follows:

```
- (IBAction)cut:(id)sender
{
    [self copy: nil];
    [self deleteSelectedExpense: nil];
}
```

Lazy Writing

The ability to provide multiple representations of data to the pasteboard is a powerful feature but can result in your application incurring considerably more overhead than is necessary. Consider a bitmap-editing application where you want to support copying of images in a variety of different formats. If in your copy method you had to create each representation, this could require a lot of processing and significant memory overhead—after which the user might decide not to paste anyway. To avoid this situation, `NSPasteboard` supports the technique of lazy writing.

If you use lazy writing, you declare the types you can supply to a pasteboard but you do not set the corresponding data. If data is subsequently requested from a pasteboard in a format that is not present, the pasteboard owner is sent a `pasteboard:provideDataForType:` message asking it to supply the data in that format. The pasteboard owner must obviously keep the original data as long as necessary to fulfill any request. Following this pattern, the copy method in the Expenses application might simply contain:

```
NSPasteboard *pb = [NSPasteboard generalPasteboard];
NSArray *types = [NSArray arrayWithObjects:
```



```
    ExpensePBoardType, NSStringPboardType, NSRTFPboardType, nil];
[pb declareTypes:types owner:self];
// cache the item to be copied, in its current state
```

You then implement a `pasteboard:provideDataForType:` method. For the Expenses application, it might look similar to the following:

```
- (void)pasteboard:(NSPasteboard *)sender provideDataForType:(NSString *)type
{
    if ([type isEqualToString:ExpensePBoardType]) {
        [sender setData:[NSKeyedArchiver archivedDataWithRootObject:cachedExpense]
            forType:ExpensePBoardType];
    }
    else if ([type isEqualToString:NSStringPboardType]) {
        [sender setString:[cachedExpense stringRepresentation]
                forType:NSStringPboardType];
    }
    else if ([type isEqualToString:NSRTFPboardType]) {
        NSAttributedString *rtfDescription = [cachedExpense rtfRepresentation];
        NSData *rtfData = [rtfDescription
                    RTFFromRange:(NSMakeRange(0, [rtfDescription length]))
                    documentAttributes:nil];
        [sender setData:rtfData forType:NSRTFPboardType];
    }
}
```

The `pasteboard:provideDataForType:` messages may also be sent to the owner when the application is shut down through an application's `terminate:` method (invoked in response to a Quit command). The user can therefore copy something, quit the application, and still paste the data that was copied.

To ensure you don't keep the cached data longer than necessary, you also need to know when the user copies something else. If the user performs another copy, the pasteboard owner is sent a `pasteboardChangedOwner:` message.

```
- (void)pasteboardChangedOwner:(NSPasteboard *)sender
{
```



```
    // remove cached expense
}
```


Named Pasteboards

Data in the pasteboard server is associated with a name (a string) that indicates how it is to be used. Each set of data and its associated name is, in effect, a separate pasteboard, distinct from the others. An application keeps a separate `NSPasteboard` object for each named pasteboard that it uses. There are five standard pasteboards in common use, each named by a global string variable:

Pasteboard Name	Description
`NSGeneralPboard`	The pasteboard that is used for ordinary cut, copy, and paste operations. It holds the contents of the last selection that has been cut or copied.
`NSFontPboard`	The pasteboard that holds font and character information and supports Copy Font and Paste Font commands that may be implemented in a text editor.
`NSRulerPboard`	The pasteboard that holds information about paragraph formats in support of the Copy Ruler and Paste Ruler commands that may be implemented in a text editor.
`NSFindPboard`	The pasteboard that holds information about the current state of the active application's Find panel. This information permits users to enter a search string into the Find panel, then switch to another application to conduct another search.
`NSDragPboard`	The pasteboard that stores data to be moved as the result of a drag operation. For additional information on working with the drag pasteboard, see Drag and Drop.

You can create private pasteboards by asking for an `NSPasteboard` object with any name other than those listed above. Data in a private pasteboard may then be shared by passing its name between applications.

The `NSPasteboard` class makes sure there is never more than one object for each named pasteboard on the computer for each user. If you ask for a new object when one has already been created for the pasteboard with that name, the existing object is returned.

Data Types

Data can be placed in the pasteboard server in more than one representation. For example, an image might be provided both in Tag Image File Format (TIFF) and as encapsulated PostScript code (EPS). Multiple representations give pasting applications the option of choosing which data type to use. In general, an application taking data from the pasteboard should choose the richest representation it can handle—rich text over plain ASCII, for example. An application putting data in the pasteboard should promise to supply it in as many data types as possible, so that as many different applications as possible can use it.

Filtering services (see "Filter Services" (page 21)) transform the data from one representation to another. Typically, these services are not invoked until data are read from a pasteboard.

Data types are identified by string objects containing the full type name. These global variables identify the string objects for the standard pasteboard types:

`NSColorPboardType`
`NSInkTextPboardType`
`NSFileContentsPboardType`
`NSFilesPromisePboardType`
`NSCreateFileContentsPboardType`
`NSCreateFilenamePboardType`
`NSFilenamesPboardType`
`NSFindPanelSearchOptionsPboardType`
`NSFontPboardType`
`NSHTMLPboardType`
`NSPDFPboardType`
`NSPICTPboardType`
`NSPostScriptPboardType`
`NSRTFPboardType`

NSRTFDPboardType
NSRulerPboardType
NSSoundPboardType
NSStringPboardType
NSTabularTextPboardType
NSTIFFPboardType
NSURLPboardType
NSVCardPboardType
WebArchivePboardType

Typically, data is written to the pasteboard using `setData:forType:` and read using `dataForType:`. Some of these types can only be written with certain methods. For instance, `NSFilenamesPboardType`'s form is an array of `NSString` objects and requires special handling. Use these methods to write these types:

Type	Writing Method	Reading Method
`NSColorPboardType`	`NSColor` class methods	`NSColor` class methods
`NSFileContents-PboardType`	`writeFileContents:`	`readFileContentsType: toFile:`
`NSFilenamesPboardType`	`setPropertyList: forType:`	`propertyListForType:`
`NSStringPboardType`	`setString:forType:`	`stringForType:`
`NSURLPboardType`	`writeToPasteboard:` (NSURL)	`URLFromPasteboard:` (NSURL)
NSFontPboardType	See "Writing Font Data" (page 18)	See "Reading Font Data" (page 19)

Types other than those listed above can also be used. For example, your application may keep data in a private format that is richer than any of the existing types. That format can also be used as a pasteboard type.

Reading and Writing Font Data

Font information is stored on the font pasteboard as an attribute of RTF data from an attributed string. To copy and paste a font, therefore, you need to respectively create an attributed string with the appropriate attribute and unpack the information from an attributed string.

Writing Font Data

You copy a font using the pasteboard named `NSFontPboard`. You must first get the pasteboard, then declare the appropriate type—`NSFontPboardType`. To create the data to place on the pasteboard, you create an instance of `NSAttributedString`; the string itself is arbitrary, but you must specify an attribute dictionary that contains the key `NSFontAttributeName` and the corresponding value the font that you want to write to the pasteboard. You then create from the string an `NSData` object to represent the RTF data and set that as the data on the pasteboard for the `NSFontPboardType`.

The following code example illustrates how to write font information to the font pasteboard. The example uses a statically-defined font; in your code you typically find the font of the currently-selected text item and use that.

```
NSPasteboard *pb = [NSPasteboard pasteboardWithName:NSFontPboard];
[pb declareTypes:[NSArray arrayWithObject:NSFontPboardType] owner:self];

NSFont *font = [NSFont fontWithName:@"Helvetica" size:12.0];
NSDictionary *attributes = [NSDictionary dictionaryWithObject:font
forKey:NSFontAttributeName];

NSAttributedString *aString = [[NSAttributedString alloc] initWithString:@"a"
attributes:attributes];
NSRange aRange = NSMakeRange(0, 1);

NSData *aStringData = [aString RTFFromRange:aRange documentAttributes:nil];
[aString release];

[pb setData:aStringData forType:NSFontPboardType];
```

Reading Font Data

You read a font using the pasteboard named `NSFontPboard`. You must first get the pasteboard, then ask the pasteboard for the appropriate type—`NSFontPboardType`. The font data is in the form of RTF data created from an attributed string. You therefore create an instance of `NSAttributedString` from this data, then get the font attribute from the attributed string.

The following code example illustrates how to read font information from the font pasteboard.

```
NSPasteboard *pb = [NSPasteboard pasteboardWithName:NSFontPboard];
NSString *bestType = [pb availableTypeFromArray:
                              [NSArray arrayWithObject:NSFontPboardType]];
NSFont *font;

if (bestType != nil) {

    NSData *data = [pb dataForType:NSFontPboardType];
    NSAttributedString *aString =
            [[NSAttributedString alloc] initWithRTF:data documentAttributes:NULL];

    if (aString != nil) {
       font = [aString attribute:NSFontAttributeName atIndex:0 effectiveRange:NULL];
        NSLog(@"font: %@", [font description]);
    }
    else {
        NSLog(@"couldn't get attributed string");
    }
    [aString release];
}
else {
    NSLog(@"couldn't get NSFontPboardType");
}
```


Reading and Writing RTFD Data

The `NSRTFDPboardType` is used for the contents of an RTFD file package (a directory containing an RTF text file and one or many image files). There are several ways to work with RTFD data. If you have an `NSFileWrapper` object that represents an RTFD file wrapper, you can send it a `serializedRepresentation` message to return the RTFD data and write that to the pasteboard as follows:

```
NSFileWrapper *tempRTFDData = [[[NSFileWrapper alloc]
    initWithPath:@"/tmp/MyTemporaryRTFDDocument.rtfd"] autorelease];
[pboard setData:[tempRTFDData serializedRepresentation]
    forType:NSRTFDPboardType];
```

In addition to `NSFileWrapper`, instances of classes such as `NSAttributedString` and `NSText` can return RTFD data. If you are using one of these classes, you can write an RTFD representation of their contents to the pasteboard as follows:

```
NSAttributedString *attrString = /* get an attributed string */;
NSRange wholeStringRange = NSMakeRange(0, [attrString length]);
NSData *rtfdData = [attrString RTFDFromRange:wholeStringRange
                                documentAttributes:nil];
[pboard setData:rtfdData forType:NSRTFDPboardType];
```

Note that the `NSText` method does not require the `documentAttributes` parameter.

Filter Services

Filter services (see System Services) provide a way to extend the types of data `NSPasteboard` can provide to applications. In addition to the data types explicitly declared for a pasteboard, you can request the data in a type to which a filter service can convert any of the declared types. Files and `NSData` objects can be converted as well using filters.

The `NSPasteboard` class uses filter services when you invoke one of the following methods:

```
+ (NSArray *)typesFilterableTo:(NSString *)type
+ (NSPasteboard *)pasteboardByFilteringFile:(NSString *)filename
+ (NSPasteboard *)pasteboardByFilteringData:(NSData *)data ofType:(NSString *)type
+ (NSPasteboard *)pasteboardByFilteringTypesInPasteboard:(NSPasteboard *)pboard
```

The first returns an array of all the data types which can be converted to `type`. The last three return pasteboards with data that is filtered into all types derivable from the current types using available filter services. Filter services are not invoked, and the data converted, until data are requested from the pasteboard, so these methods are reasonably inexpensive.

Because filter services commonly translate data from unknown file formats into known formats, you need a way of dynamically specifying pasteboard types. The filter services and pasteboard facilities define types based on file extensions and HFS file types with these functions:

```
NSString *NSCreateFilenamePboardType(NSString *fileType)
NSString *NSCreateFileContentsPboardType(NSString *fileType)
NSString *NSGetFileType(NSString *pboardType)
NSArray *NSGetFileTypes(NSArray *pboardTypes)
NSString *NSFileTypeForHFSTypeCode(OSType hfsFileTypeCode)
OSType NSHFSTypeCodeFromFileType(NSString *fileTypeString)
```

The `fileType` argument is either a file extension, minus the period (for example, "`eps`" or "`tiff`"), or an HFS file type encoded with the `NSFileTypeForHFSTypeCode` function (for example, "`'TEXT'`" or "`'MooV'`"). You create pasteboard type strings with the first two functions, and get file types (extensions or encoded HFS types) from pasteboard type strings with the second two functions. The last two functions convert between HFS file types (`OSType`) and encoded HFS type strings.

Providing a Filter Service

You implement a filter service very much like a system service. See System Services for details on how services generally work. The following sections focus on issues specific to filter services.

Creating the Filter

Like system services, filter services are defined with an `NSServices` property in the filter's information property list file (`Info.plist`). Filter services, though, do not show up in the Services menu, so you do not need to have `NSMenuItem` and `NSKeyEquivalent` entries in the definition.

Because data is moving both in and out of the filter service, you must have entries for both `NSSendTypes` and `NSReturnTypes` in the filter definition. You indicate send and return types as either `NSTypedFilenamesPboardType:fileType` when you want file names and `NSTypedFileContentsPboardType:fileType` when you want file contents, where `fileType` is either a file name extension or an encoded HFS type, for example:

```
NSSendTypes = (NSTypedFilenamesPboardType:tiff);
NSSendTypes = (NSTypedFileContentsPboardType:'MooV');
```

Finally, instead of an NSMessage entry, which identifies the method to invoke, filter services contain an equivalent `NSFilter` entry. The invoked method is `filterName:userData:error:`, where `filterName` is the value of the `NSFilter` entry. The method accepts a pasteboard, converts the contents of the pasteboard to the requested type or types, and returns the converted data on the pasteboard.

The method identified by the `NSFilter` property is sent to the filter application's service provider object, which you register with the pasteboard server using the function `NSRegisterServicesProvider` when the filter service is launched. This function's declaration is:

```
void NSRegisterServicesProvider(id provider, NSString *name)
```

`provider` is the object that provides the services, and `name` is the same value you specify for the `NSPortName` entry in the services specification. `NSPortName` is usually the filter application's name. After calling `NSRegisterServicesProvider`, the filter service must enter the run loop to respond to service requests. The filter's `main` function may look like this:

```
int main (int argc, const char *argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    ServiceTest *serviceProvider = [[ServiceTest alloc] init];

    NSRegisterServicesProvider(serviceProvider, @"SimpleService");

    NS_DURING
        [[NSRunLoop currentRunLoop] run];
    NS_HANDLER
        NSLog(@"Received exception: %@", localException);
    NS_ENDHANDLER

    [serviceProvider release];
    [pool release];
    return 0;
}
```

If the `serviceProvider` object implements the method `convertData:userData:error:`, the filter's `Info.plist` file may contain the following service specification:

```
NSServices = (
    {
        NSFilter = "convertData";
        NSPortName = SimpleService;
        NSSendTypes = (NSTypedFilenamesPboardType:gif);
        NSReturnTypes = (NSTIFFPboardType);
    }
);
```


The filter service bundle should have a `.service` extension and be installed in the `Library/Services` directory in one of the file system domains—System, Network, Local, or User. (See *System Overview* for details on file-system domains.) The list of available services is created each time a user logs into the computer, so you must log out and back in before a newly-installed service is available.

Using Alternate Input Mechanisms

A filter service can use data-transfer mechanisms other than the pasteboard, indicated by an optional entry in the filter service specification. The key is `NSInputMechanism`, and it can have a value of `NSUnixStdio`, `NSMapFile`, or `NSIdentity`. If you specify an input mechanism, the value for the `NSFilter` entry is ignored (though it is still required).

NSUnixStdio

`NSUnixStdio` allows you to turn nearly any command-line program into a filter service. Instead of sending an Objective-C message to an object in your filter service program, the services facility simply runs the executable specified in the service specification with the contents of the pasteboard as the argument (which must be of `NSFilenamesPboardType` or `NSTypedFilenamesPboardType`). If there is more than one filename on the pasteboard, only the first is used. The output of the filter program (on `stdout`) is captured by the services facility and put on a pasteboard for use by the requestor of the filter. Note that the program must be relaunched every time the service is invoked; if you are creating a filter service from scratch it is more efficient to package it as an application that can remain running. Here is a sample service specification for a program that converts GIF images to TIFF:

```
NSServices = (
    {
        NSFilter = "";
        NSPortName = gif2tiff;
        NSInputMechanism = NSUnixStdio;
        NSSendTypes = (NSTypedFilenamesPboardType:gif);
        NSReturnTypes = (NSTIFFPboardType);
    }
);
```

NSMapFile

`NSMapFile` defines an "empty" service for data in files, used when you invoke NSPasteboard's `pasteboardByFilteringFile:` class method. Its value must be an `NSFilenamesPboardType` or an `NSTypedFilenamesPboardType`. When the filter service is invoked for a file, the services facility merely puts the contents of the file on the pasteboard. This input mechanism is useful for file types with nonstandard or special extensions whose format is nonetheless the same as a standard type. For example, if you have defined an image format based on a subset of TIFF and given it a file extension of `stif`, you can define a service that maps the `stif` file extension to `NSTIFFPboardType`:

```
NSServices = (
    {
        NSFilter = "";
        NSInputMechanism = NSMapFile;
        NSSendTypes = (NSTypedFilenamesPboardType:stif);
        NSReturnTypes = (NSTIFFPboardType);
    }
);
```

`NSMapFile` does not result in any program being executed, so its service specification lacks the `NSPortName` entry.

NSIdentity

`NSIdentity` defines an empty service for data in memory, used when you invoke NSPasteboard's `pasteboardByFilteringData:ofType:` class method. It declares that the send type is effectively identical to the return type—though the reverse is not necessarily true. For example, you can define a service that filters your custom image format in memory with this service specification:

```
NSServices = (
    {
        NSFilter = "";
        NSInputMechanism = NSIdentity;
        NSSendTypes = (MyCustomImagePboardType);
        NSReturnTypes = (NSTIFFPboardType);
    }
);
```

`NSIdentity` does not result in any program being executed, so its service specification lacks the `NSPortName` entry.

Document Revision History

This table describes the changes to *Pasteboard Programming Topics for Cocoa*.

Date	Notes
2009-01-20	Added note to direct readers to new guide for Mac OS X v10.6 and later.
2007-07-13	Updated for Mac OS X v10.5.
2006-05-23	Changed title from Copying and Pasting. Added new articles on implementing copy and paste and reading and writing font data.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

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