Device Features Programming Guide

Data Management: Device Information

2010-04-30

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

About Integrating Device Features into Your Application

Creating a best-of-class mobile application includes taking advantage of device hardware and software features that support the intent of your application. In iOS devices, these features range from the built-in calendar and address book, to the iPod library and audio hardware, to the camera, layered maps, and location awareness. This document surveys these and other features and gives you programming guidance on integrating them into your app.

In some cases, a feature is fully described in its own programming guide. In other cases, this document introduces you to the feature and provides code examples. Be sure to read the next chapter, "A Survey of Device Features" (page 9), to learn about the range of device features available to you and where to find out more about them.

Prerequisites

Before reading this document, you should be familiar with Cocoa Touch development as introduced in *iOS Application Programming Guide*. You should also be comfortable with Cocoa features, concepts, and terminology as described in *Cocoa Fundamentals Guide*.

Organization of This Document

This document includes the following chapters:

- "A Survey of Device Features" (page 9) briefly describes the range of hardware and software-based device features you can use in your app.
- "Using the Camera and Photo Library" (page 13) describes how to employ the built-in camera and how to access the user's Photo library.
- "Previewing and Opening Items" (page 17) explains how to provide document and image previews, and how to ask the system to find an application to open items that your application doesn't support.
- "Using Copy, Cut, and Paste Operations" (page 23) describes the ways you can support basic editing features in your app.
- "Adding Custom Edit Menu Items" (page 33) describes how to add your own menu item to the system editing menu.
- "Using System Messaging Facilities" (page 35) explains how to add email and SMS composition and sending to your app.
- "Accessing Cellular Telephone Information" (page 39) introduces the Core Telephony framework, which you can use to support a VoIP (Voice over Internet Protocol) app.

See Also

The dedicated programming guides that describe device features include:

- *Address Book Programming Guide for iOS* describes how to access the user's contacts.
- *Event Kit Programming Guide* describes how to access the user's calendar.
- *Multimedia Programming Guide* describes how to use audiovisual hardware and software features.
- *Location Awareness Programming Guide* describes how to access the user's location and how to use maps.
- *External Accessory Programming Topics* describes how to interact with external hardware accessories.

Availability

Many device features described in this document have been available since iOS 2.0. Others, such as Quick Look, SMS support, and Calendar integration were first available in iOS 4.0. Refer to the iOS Reference Library for detailed availability information.

A Survey of Device Features

Your iPhone application does not need to be isolated from the rest of the system. In fact, the best applications take advantage of hardware and software features built in to the device to provide a richer, more compelling experience for the user.

This brief chapter surveys the various device features you may want to use. In some cases, a feature is described in detail in a later chapter in this document. In other cases, the feature has a dedicated programming guide.

Address Book and Contacts

To access the user's contacts, use the Address Book framework. You can also present standard system interfaces for picking and creating contacts with the Address Book UI framework.

For details, see *Address Book Programming Guide for iOS*.

Calendar Events

To schedule and access time-based events, use the Event Kit and Event Kit UI frameworks. Events scheduled using the Event Kit framework show up in the Calendar application and in other applications that support that framework.

For details, see *Event Kit Programming Guide*, *Event Kit Framework Reference*, and *Event Kit UI Framework Reference*.

Maps and Location

To incorporate maps into your application, and to add information layers on top of those maps, use the Map Kit framework.

To take advantage of the user's location—such as to limit a search for restaurants to those within a specified radius—use the Core Location framework.

For details on each of these features, see *Location Awareness Programming Guide*.

Camera and Photo Library

To use the camera for taking pictures or movies, or to access the user's photo library, use the `UIImagePickerController` class.

For details, see "Using the Camera and Photo Library" (page 13).

For more advanced use of the camera, including adding augmented reality features to your application, refer to *AV Foundation Framework Reference*.

Audio Hardware and iPod Library

To access the contents of a user's iPod Library, use the Media Player framework. This framework lets you create your own music player that has access to all of the audio contents of the user's iPod Library. You can also use this framework to play items from the iPod Library while your game or other application is running.

Beyond this, iOS devices provide a wide range of audio hardware and software features you can take advantage of, among them the following:

- Microphone
- Speakers
- Hardware and software codecs
- Audio processing plug-ins (*audio units*)

To access these features, use the various Core Audio frameworks and the audio-specific classes in the AV Foundation framework.

For details, see "Using Audio" in *Multimedia Programming Guide*.

Item Quick Look Previews

To provide previews of iWork documents, PDF files, images, and other items, use the `UIDocumentInteractionController` class or the Quick Look framework. A document interaction controller can also help a user find an application on their system that is capable of opening an item. It does this by way of an options menu in the user interface.

Applications most likely to benefit from previews are those which may receive items that the application cannot open directly. For example, if you are writing an email agent or a remote disk browser, you will likely want to include preview support.

For details, see "Previewing and Opening Items" (page 17).

Copy, Cut, and Paste Operations

iOS supports copy, cut, and paste operations within and between applications. You gain access to these features by using the `UITextView`, `UITextField`, and `UIWebView` classes, or by implementing copy/cut/paste in your application using various UIKit classes.

For details, see "Using Copy, Cut, and Paste Operations" (page 23).

Edit Menu

iOS provides a context-sensitive edit menu that can display the system commands Copy, Cut, Paste, Select, Select All, and Delete. To add a custom item to this menu, create an instance of the `UIMenuItem` class and add it to the singleton `UIMenuController` object.

For details, see "Adding Custom Edit Menu Items" (page 33).

Custom Input and Accessory Views

You can substitute a custom input view for the system keyboard. You can also enhance the system keyboard with an accessory view for customized input or to provide information to the user. To use this facility, set the `inputView` or `inputAccessoryView` properties of a `UITextView` object (or any object that inherits from the `UIResponder` class).

For details, see *UIResponder Class Reference*.

Mail and Messages (SMS)

To present the standard system interfaces for composing and sending email or SMS messages, use the view controller classes in the Message UI framework.

For details, see "Using System Messaging Facilities" (page 35).

Cellular Telephone Information

To access information about active cellular telephone calls, or to access cellular service provider information from the user's SIM card, use the Core Telephony framework. VoIP (Voice over Internet Protocol) applications are likely to benefit from these features.

For details, see "Accessing Cellular Telephone Information" (page 39).

User Interaction Event Handling

User interaction events are objects that inform your application of user actions—such as when a user touches a view, tilts the device, or presses a button on the headset. To work with events, including complex gesture events such as multitouch and shaking, use the `UIEvent`, `UIAccelerometer`, and related classes in the UIKit framework.

For details, see *Event Handling Guide for iOS*.

Hardware Accessories

To interact with external hardware connected by wire or Bluetooth, use the External Accessory framework.

For details, see *External Accessory Programming Topics*.

Using the Camera and Photo Library

Note: This chapter contains information that used to be in *iOS Application Programming Guide*. This information has not been updated specifically for iOS 4.0.

Taking Pictures with the Camera

The UIKit framework provides access to a device's camera through the `UIImagePickerController` class. An instance of this class, called an *image picker controller*, displays the standard system interface for taking pictures using the camera. You can configure it to display optional controls for resizing and cropping a picture that the user has just taken. In addition, you can use an image picker controller to let the user pick and display photos from their photo library.

If a device doesn't have a camera available, you cannot display the camera interface. Before attempting to display it, check whether the camera is available by calling the `isSourceTypeAvailable:` class method of the `UIImagePickerController` class. Always respect this method's return value. If this method returns `NO`, the current device does not have a camera or the camera is currently unavailable for some reason. If the method returns `YES`, you can display the picture taking interface as follows:

1. Create a new `UIImagePickerController` object.
2. Assign a delegate object to the image picker controller.

   Typically, you'd use the current view controller as the delegate for the picker, but you can use a different object if you prefer. The delegate object must conform to the `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate`protocols.

   Note: If your delegate does not conform to the `UINavigationControllerDelegate` protocol, you may see a warning during compilation. However, because the methods of this protocol are optional, the warning has no impact on your code. To eliminate the warning, include the `UINavigationControllerDelegate` protocol in the list of supported protocols for your delegate's class. You do not need to implement the corresponding methods.

3. Set the picker type to `UIImagePickerControllerSourceTypeCamera`.
4. Optionally, enable or disable the picture editing controls by assigning an appropriate value to the `allowsEditing`property.
5. Call the `presentModalViewController:animated:` method of the currently active view controller, passing a `UIImagePickerController` object as the new view controller.

   The image picker controller slides the camera interface into position, where it remains active until the user approves the picture or cancels the operation. At that time, the picker controller notifies its delegate of the user's choice.

Note: Never directly access the view that is managed by an image picker controller. Instead, use the controller's methods.

Listing 2-1 shows typical code for implementing the preceding set of steps. As soon as you call the `presentModalViewController:animated` method, the picker controller takes over, displaying the camera interface and responding to all user interactions until the interface is dismissed. To let the user choose a photo from their photo library (instead of take a picture), change the value in the `sourceType` property of the picker to `UIImagePickerControllerSourceTypePhotoLibrary`.

Listing 2-1 Displaying the interface for taking pictures

```
-(BOOL)startCameraPickerFromViewController:(UIViewController*)controller
usingDelegate:(id<UIImagePickerControllerDelegate>)delegateObject
{
    if ( (![UIImagePickerController
isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera])
            || (delegateObject == nil) || (controller == nil))
        return NO;

    UIImagePickerController* picker = [[UIImagePickerController alloc] init];
    picker.sourceType = UIImagePickerControllerSourceTypeCamera;
    picker.delegate = delegateObject;
    picker.allowsEditing = YES;

    // Picker is displayed asynchronously.
    [controller presentModalViewController:picker animated:YES];
    return YES;
}
```

When the user taps the appropriate button to dismiss the camera interface, the `UIImagePickerController` notifies the delegate of the user's choice but does not dismiss the interface. The delegate is responsible for dismissing the picker interface. (Your application is also responsible for releasing the picker when done with it, which you can do in the delegate methods.) It is for this reason that the delegate is actually the view controller object that presented the picker in the first place. Upon receiving the delegate message, the view controller would call its `dismissModalViewControllerAnimated:` method to dismiss the camera interface.

Listing 2-2 shows the delegate methods for dismissing the camera interface displayed in Listing 2-1 (page 14). These methods are implemented by a custom `MyViewController` class, which is a subclass of `UIViewController` and, for this example, is considered to be the same object that displayed the picker in the first place. The `useImage:` method is an empty placeholder for the work you would do in your own version of this class and should be replaced by your own custom code.

Listing 2-2 Delegate methods for the image picker

```
@implementation MyViewController (ImagePickerDelegateMethods)

- (void)imagePickerController:(UIImagePickerController *)picker
                didFinishPickingImage:(UIImage *)image
                editingInfo:(NSDictionary *)editingInfo
{
    [self useImage:image];

    // Remove the picker interface and release the picker object.
    [[picker parentViewController] dismissModalViewControllerAnimated:YES];
    [picker release];
```

```
}

- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker
{
    [[picker parentViewController] dismissModalViewControllerAnimated:YES];
    [picker release];
}

// Implement this method in your code to do something with the image.
- (void)useImage:(UIImage*)theImage
{
}
@end
```

If image editing is enabled and the user successfully picks an image, the `image` parameter of the `imagePickerController:didFinishPickingImage:editingInfo:` method contains the edited image. You should treat this image as the selected image, but if you want to store the original image, you can get it (along with the crop rectangle) from the dictionary in the `editingInfo` parameter.

Recording and Editing Video

Starting in iOS 3.0, you can record video, with included audio, on supported devices. To display the video recording interface, create and push a `UIImagePickerController` object, just as for displaying the still-camera interface.

To record video, you must first check that the camera source type (`UIImagePickerControllerSourceTypeCamera`) is available and that the movie media type `kUTTypeMovie` is available for the camera. Depending on the media types you assign to the `mediaTypes` property, the picker can directly display the still camera or the video camera, or a selection interface that lets the user choose.

Using the `UIImagePickerControllerDelegate` protocol, register as a delegate of the image picker. Your delegate object receives a completed video recording by way of the `imagePickerController:didFinishPickingMediaWithInfo:` method.

On supported devices, you can also pick previously-recorded videos from a user's photo library.

For more information on using the image picker class, see *UIImagePickerController Class Reference*. For information on trimming recorded videos, see *UIVideoEditorController Class Reference* and *UIVideoEditorControllerDelegate Protocol Reference*.

Picking an Item from the Photo Library

UIKit provides access to the user's photo library through the `UIImagePickerController` class. This controller displays a photo picker interface, which provides controls for navigating the user's photo library and selecting an image to return to your application. You also have the option of enabling user editing controls, which let the user the pan and crop the returned image. This class can also be used to present a camera interface.

Because the `UIImagePickerController` class is used to display the interface for both the camera and the user's photo library, the steps for using the class are almost identical for both. The only difference is that you assign the `UIImagePickerControllerSourceTypePhotoLibrary` value to the `sourceType` property of the picker object. The steps for displaying the camera picker are discussed in "Taking Pictures with the Camera" (page 13).

Note: As you do for the camera picker, you should always call the `isSourceTypeAvailable:` class method of the `UIImagePickerController` class and respect the return value of the method. You should never assume that a given device has a photo library. Even if the device has a library, this method could still return `NO` if the library is currently unavailable.

Previewing and Opening Items

Your application can provide previews of items that you otherwise could not open—iWork documents, PDF files, images, and others. To do this, use an instance of the `UIDocumentInteractionController` class, which also lets you open a previewed item in an appropriate application, if one is available.

If your application can display specific file types, you can register that capability in your Xcode project's `Info.plist` file. When another application asks the system for help opening a previewed item of those types, your application will be included in the options menu presented to the user.

Previewing and Opening Items

When your application needs to interact with certain types of items that it cannot open on its own, you can use a `UIDocumentInteractionController` object to manage those interactions. A document interaction controller works with the Quick Look framework to determine whether files can be previewed in place or opened by another application. Your application, in turn, works with the document interaction controller to present the available options to the user at appropriate times.

To use a document interaction controller in your application, do the following:

1. Create an instance of the `UIDocumentInteractionController` class for each file you want to manage.
2. Present the file in your application's user interface. (Typically, you would do this by displaying the file name or icon somewhere in your interface.)
3. When the user interacts with the file, ask the document interaction controller to present one of the following interfaces:
    - A file preview view that displays the contents of the file
    - A menu containing options to preview the file, copy its contents, or open it using another application
    - A menu prompting the user to open it with another application

Any application that interacts with files can use a document interaction controller. Programs that download files from the network are the most likely candidates to need these capabilities. For example, an email program might use document interaction controllers to preview or open files attached to an email. Of course, you do not need to download files from the network to use this feature. If your application supports file sharing, you might need to use a document interaction controller to process files of unknown types discovered in your application's `Documents/Shared` directory.

Creating and Configuring a Document Interaction Controller

To create a new document interaction controller, initialize a new instance of the `UIDocumentInteractionController` class with the file you want it to manage and assign a delegate object. The delegate is responsible for providing the document interaction controller with information it needs to present its views. You can also use the delegate to perform additional actions when those views are displayed. The following code creates a new document interaction controller and sets the delegate to the current object. Note that the caller of this method needs to retain the returned object.

```
- (UIDocumentInteractionController*)docControllerForFile:(NSURL)fileURL
{
   UIDocumentInteractionController* docController =
       [UIDocumentInteractionController interactionControllerWithURL:fileURL];
   docController.delegate = self;

   return docController;
}
```

Once you have a document interaction controller, you can use its properties to get information about the file, including its name, type information, and path information. The controller also has an `icons` property that contains `UIImage` objects representing the document's icon in various sizes. You can use all of this information when presenting the document in your user interface.

If you plan to let the user open the file in another application, you can use the `annotation` property of the document interaction controller to pass custom information to the opening application. It is up to you to provide information in a format that the other application will recognize. For example, this property is typically used by application suites that want to communicate additional information about a file to other applications in the suite. The opening application sees the annotation data in the `UIApplicationLaunchOptionsAnnotationKey` key of the options dictionary that is passed to it at launch time.

Presenting a Document Interaction Controller

When the user interacts with a file, you use the document interaction controller to display the appropriate user interface. You have the choice of displaying a document preview or of prompting the user to choose an appropriate action for the file using one of the following methods:

- Use the `presentOptionsMenuFromRect:inView:animated:` or `presentOptionsMenuFromBarButtonItem:animated:` method to present the user with a variety of options.
- Use the `presentPreviewAnimated:` method to display a document preview.
- Use the `presentOpenInMenuFromRect:inView:animated:` or `presentOpenInMenuFromBarButtonItem:animated:` method to present the user with a list of applications with which to open the file.

Each of the preceding methods attempts to display a custom view with the appropriate content. When calling these methods, you should always check the return value to see if the attempt was actually successful. These methods may return `NO` if the resulting interface would have contained no content. For example, the `presentOpenInMenuFromRect:inView:animated:` method returns `NO` if there are no applications capable of opening the file.

If you choose a method that might display a preview of the file, your delegate object must implement the `documentInteractionControllerViewControllerForPreview:` method. Document previews are displayed using a modal view, so the view controller you return becomes the parent of the modal document preview. If you do not implement this method, if your implementation returns `nil`, or if the specified view controller is unable to present another modal view controller, a document preview is not displayed.

Normally, the document interaction controller automatically handles the dismissal of the view it presents. However, you can dismiss the view programmatically as needed by calling the `dismissMenuAnimated:` or `dismissPreviewAnimated:` methods.

Registering the File Types Your Application Supports

If your application is capable of opening specific types of files, you should register that support with the system. To declare its support for file types, your application must include the `CFBundleDocumentTypes` key in its `Info.plist`file. The system gathers this information from your application and maintains a registry that other applications can access through a document interaction controller.

The `CFBundleDocumentTypes` key contains an array of dictionaries, each of which identifies information about a specific document type. A document type usually has a one-to-one correspondence with a particular document type. However, if your application treats more than one file type the same way, you can group those types together as a single document type. For example, if you have two different file formats for your application's native document type, you could group both the old type and new type together in a single document type entry. By doing so, both the new and old files would appear to be the same type of file and would be treated in the same way.

Each dictionary in the `CFBundleDocumentTypes` array can include the following keys:

- `CFBundleTypeName` specifies the name of the document type.
- `CFBundleTypeIconFiles` is an array of filenames for the image resources to use as the document's icon.
- `LSItemContentTypes` contains an array of strings with the UTI types that represent the supported file types in this group.
- `LSHandlerRank` describes whether this application owns the document type or is merely able to open it.

From the perspective of your application, a document is a file type (or file types) that the application supports and treats as a single entity. For example, an image processing application might treat different image file formats as different document types so that it can fine tune the behavior associated with each one. Conversely, a word processing application might not care about the underlying image formats and just manage all image formats using a single document type.

Listing 3-1 shows a sample XML snippet from the `Info.plist` of an application that is capable of opening a custom file type. The `LSItemContentTypes` key identifies the UTI associated with the file format and the `CFBundleTypeIconFiles` key points to the icon resources to use when displaying it.

Listing 3-1 Document type information for a custom file format

```
<dict>
   <key>CFBundleTypeName</key>
   <string>My File Format</string>
```

```
    <key>CFBundleTypeIconFiles</key>
        <array>
            <string>MySmallIcon.png</string>
            <string>MyLargeIcon.png</string>
        </array>
    <key>LSItemContentTypes</key>
        <array>
            <string>com.example.myformat</string>
        </array>
    <key>LSHandlerRank</key>
    <string>Owner</string>
</dict>
```

For more information about the contents of the `CFBundleDocumentTypes` key, see the description of that key in *Information Property List Key Reference*.

Opening Supported File Types

At launch time, the system may ask your application to open a specific file and present it to the user. This typically occurs because another application encountered the file and used a document interaction controller to handle it. You receive information about the file to be opened in the `application:didFinishLaunchingWithOptions:` method of your application delegate. If your application handles custom file types, you must implement this delegate method (instead of the `applicationDidFinishLaunching:` method) and use it to initialize your application.

The options dictionary passed to the `application:didFinishLaunchingWithOptions:` method contains information about the file to be opened. Specifically, your application should look in this dictionary for the following keys:

- `UIApplicationLaunchOptionsURLKey` contains an `NSURL` object that specifies the file to open.
- `UIApplicationLaunchOptionsSourceApplicationKey` contains an `NSString` with the bundle identifier of the application that initiated the open request.
- `UIApplicationLaunchOptionsSourceApplicationKey` contains a property list object that the source application wanted to associate with the file when it was opened.

If the `UIApplicationLaunchOptionsURLKey` key is present, your application must open the file referenced by that key and present its contents immediately. You can use the other keys in the dictionary to gather information about the circumstances surrounding the opening of the file.

Using the Quick Look Framework

To gain even more control over your previews, you can use the Quick Look framework directly. The framework's primary class is `QLPreviewController`. It relies on a delegate for responding to preview actions, and a data source for providing the preview items.

An instance of the `QLPreviewController` class, or Quick Look preview controller, provides a specialized view for previewing an item. To display a Quick Look preview controller you have two options: You can push it into view using a `UINavigationController` object, or can present it modally, full screen, using the `presentModalViewController:animated:` method of its parent class, `UIViewController`.

Choose the display option that best fits the visual and navigation style of your application. Modal, full-screen display might work best if your app doesn't use a navigation bar. If your app uses iPhone-style navigation, you might want to opt for pushing your preview into view.

A displayed preview includes a title taken from the last path component of the item URL. You can override this by implementing a `previewItemTitle` accessor for the preview item.

A Quick Look preview controller can display previews for the following items:

- iWork documents
- Microsoft Office documents (Office '97 and newer)
- Rich Text Format (RTF) documents
- PDF files
- Images
- Text files whose uniform type identifier (UTI) conforms to the `public.text` type (see *Uniform Type Identifiers Reference*)
- Comma-separated value (csv) files

To use a Quick Look preview controller, you must provide a data source object using the methods described in *QLPreviewControllerDataSource Protocol Reference*. The data source provides preview items to the controller and tells it how many items to include in a preview navigation list. If there is more than one item in the list, a modally-presented (that is, full-screen) controller displays navigation arrows to let the user switch among the items. For a Quick Look preview controller pushed using a navigation controller, you can provide buttons in the navigation bar for moving through the preview-item list.

For a complete description of the Quick Look framework, see *Quick Look Framework Reference*.

Using Copy, Cut, and Paste Operations

Note: This chapter contains information that used to be in *iOS Application Programming Guide*. This information has not been updated specifically for iOS 4.0.

Beginning with iOS 3.0, users can now copy text, images, or other data in one application and paste that data to another location within the same application or in a different application. You can, for example, copy a person's address in an email message and paste it into the appropriate field in the Contacts application. The UIKit framework currently implements copy-cut-paste in the `UITextView`, `UITextField`, and `UIWebView` classes. If you want this behavior in your own applications, you can either use objects of these classes or implement copy-cut-paste yourself.

The following sections describe the programmatic interfaces of the UIKit that you use for copy, cut, and paste operations and explain how they are used.

Note: For usage guidelines related to copy and paste operations, see "Supporting Copy and Paste" in *iPhone Human Interface Guidelines*.

UIKit Facilities for Copy-Paste Operations

Several classes and an informal protocol of the UIKit framework give you the methods and mechanisms you need to implement copy, cut, and paste operations in your application:

- The `UIPasteboard` class provides pasteboards: protected areas for sharing data within an application or between applications. The class offers methods for writing and reading items of data to and from a pasteboard.
- The `UIMenuController` class displays an editing menu above or below the selection to be copied, cut, or pasted into. The commands of the editing menu are (potentially) Copy, Cut, Paste, Select, and Select All.
- The `UIResponder` class declares the method `canPerformAction:withSender:`. Responder classes can implement this method to show and remove commands of the editing menu based on the current context.
- The `UIResponderStandardEditActions` informal protocol declares the interface for handling copy, cut, paste, select, and select-all commands. When users tap one of the commands in the editing menu, the corresponding `UIResponderStandardEditActions` method is invoked.

Pasteboard Concepts

A pasteboard is a standardized mechanism for exchanging data within applications or between applications. The most familiar use for pasteboards is handling copy, cut, and paste operations:

- When a user selects data in an application and chooses the Copy (or Cut) menu command, the selected data is placed onto a pasteboard.
- When the user chooses the Paste menu command (either in the same or a different application), the data on a pasteboard is copied to the current application from the pasteboard.

In iOS, a pasteboard is also used to support Find operations. Additionally, you may use pasteboards to transfer data between applications using custom URL schemes instead of copy, cut, and paste commands; see "Communicating with Other Applications" in *iOS Application Programming Guide* for information about this technique.

Regardless of the operation, the basic tasks you perform with a pasteboard object are to write data to a pasteboard and to read data from a pasteboard. Although these tasks are conceptually simple, they mask a number of important details. The main complexity is that there may be a number of ways to represent data, and this complexity leads to considerations of efficiency. These and other issues are discussed in the following sections.

Named Pasteboards

Pasteboards may be public or private. Public pasteboard are called **system pasteboards**; private pasteboards are created by applications, and hence are called **application pasteboards**. Pasteboards must have unique names. `UIPasteboard` defines two system pasteboards, each with its own name and purpose:

- `UIPasteboardNameGeneral` is for cut, copy, and paste operations involving a wide range of data types. You can obtain a singleton object representing the General pasteboard by invoking the `generalPasteboard` class method.
- `UIPasteboardNameFind` is for search operations. The string currently typed by the user in the search bar (`UISearchBar`) is written to this pasteboard, and thus can be shared between applications. You can obtain an object representing the Find pasteboard by calling the `pasteboardWithName:create:` class method, passing in `UIPasteboardNameFind` for the name.

Typically you use one of the system-defined pasteboards, but if necessary you can create your own application pasteboard using `pasteboardWithName:create:` If you invoke `pasteboardWithUniqueName`, `UIPasteboard` gives you a uniquely-named application pasteboard. You can discover the name of a pasteboard through its `name`property.

Pasteboard Persistence

Pasteboards can be marked persistent so that they continue to exist beyond the termination of applications that use them. Pasteboards that aren't persistent are removed when the application that created them quits. System pasteboards are persistent. Application pasteboards by default are not persistent. To make an application pasteboard persistent, set its `persistent` property to `YES`. A persistent application pasteboard is removed when a user uninstalls the owning application.

Pasteboard Owner and Items

The object that last put data onto the pasteboard is referred to as the pasteboard **owner**. Each piece of data placed onto a pasteboard is considered a pasteboard **item**. The pasteboard can hold single or multiple items. Applications can place or retrieve as many items as they wish. For example, say a user selection in a view contains both text and an image. The pasteboard lets you copy the text and the image to the pasteboard as separate items. An application reading multiple items from a pasteboard can choose to take only those items that it supports (the text, but not the image, for example).

Important: When an application writes data to a pasteboard, even if it is just a single item, that data replaces the current contents of the pasteboard. Although you may use the `addItems:` method of `UIPasteboard` to append items, the write methods of the class do not append items to the current contents of the pasteboard.

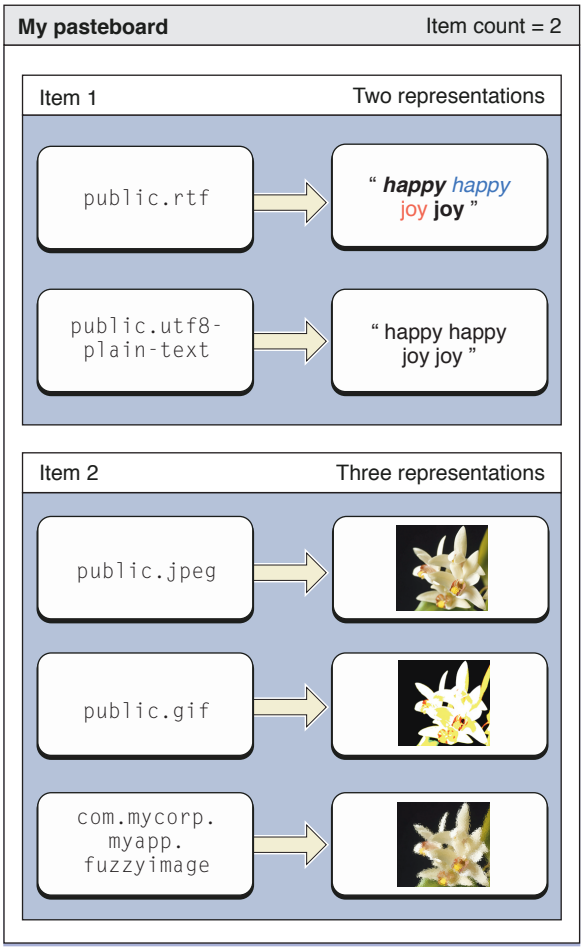
Representations and UTIs

Pasteboard operations are often carried out between two different applications. Neither application is required to know about the other, including the kinds of data it can handle. To maximize the potential for sharing, a pasteboard can hold multiple **representations** of the same pasteboard item. For example, a rich text editor might provide HTML, PDF, and plain-text representations of the copied data. An item on a pasteboard includes all representations of that data item that the application can provide.

Each representation of a pasteboard item is typically identified by a Unique Type Identifier (UTI). (A UTI is simply a string that uniquely identifies a particular data type. The UTI provides a common means to identify data types. If you have a custom data type you wish to support, you must create a unique identifier for it. For this, you could use reverse-DNS notation for your representation-type string to ensure uniqueness; for example, a custom representation type could be `com.myCompany.myApp.myType`. For more information on UTIs, see *Uniform Type Identifiers Overview*.

For example, suppose an application supported selection of rich text and images. It may want to place on a pasteboard both rich text and Unicode versions of a text selection and different representations of an image selection. Each representation of each item is stored with its own data, as shown in Figure 4-1.

Figure 4-1 Pasteboard items and representations



In general, to maximize the potential for sharing, pasteboard items should include as many different representations as possible.

A pasteboard reader must find the data type that best suits its capabilities (if any). Typically, this means selecting the richest type available. For example, a text editor might provide HTML (rich text) and plain-text representations of copied text data. An application that supports rich text should retrieve the HTML representation and an application that only supports plain text should retrieve the plain-text version.

Change Count

The change count is a per-pasteboard variable that increments every time the contents of the pasteboard changes—specifically, when items are added, modified, or removed. By examining the change count (through the `changeCount` property), an application can determine whether the current data in the pasteboard is the same as the data it last received. Every time the change count is incremented, the pasteboard sends a notification to interested observers.

Selection and Menu Management

To copy or cut something in a view, that "something" must be selected. It can be a range of text, an image, a URL, a color, or any other representation of data, including custom objects. To implement copy-and-paste behavior in your custom view, you must manage the selection of objects in that view yourself. If the user selects an object in the view by making a certain touch gesture (for example, a double-tap) you must handle that event, internally record the selection (and deselect any previous selection), and perhaps visually indicate the new selection in the view. If it is possible for users to select multiple objects in your view for copy-cut-paste operations, you must implement that multiple-selection behavior.

Note: Touch events and techniques for handling them are discussed in "Touch Events" in *iOS Application Programming Guide*.

When your application determines that the user has requested the editing menu—which could be the action of making a selection—you should complete the following steps to display the menu:

1. Call the `sharedMenuController` class method of `UIMenuController` to get the global menu-controller instance.
2. Compute the boundaries of the selection and with the resulting rectangle call the `setTargetRect:inView:` method. The editing menu is displayed above or below this rectangle, depending how close the selection is to the top or bottom of the screen.
3. Call the `setMenuVisible:animated:` method (with `YES` for both arguments) to animate the display of the editing menu above or below the selection.

Listing 4-1 illustrates how you might display the edit menu in an implementation of the `touchesEnded:withEvent:` method. (Note that the example omits the section of code that handles the selection.) This code snippet also shows the custom view sending itself a `becomeFirstResponder` message to ensure that it is the first responder for the subsequent copy, cut, and paste operations.

Listing 4-1 Displaying the editing menu

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *theTouch = [touches anyObject];

    if ([theTouch tapCount] == 2  && [self becomeFirstResponder]) {

        // selection management code goes here...

        // bring up editing menu.
        UIMenuController *theMenu = [UIMenuController sharedMenuController];
       CGRect selectionRect = CGRectMake (currentSelection.x, currentSelection.y,
 SIDE, SIDE);
        [theMenu setTargetRect:selectionRect inView:self];
        [theMenu setMenuVisible:YES animated:YES];

    }
}
```

The menu initially includes all commands for which the first responder has corresponding `UIResponderStandardEditActions` method implementations (`copy:`, `paste:`, and so on). Before the menu is displayed, however, the system sends a `canPerformAction:withSender:` message to the first responder, which in many cases is the custom view itself. In its implementation of this method, the responder evaluates whether the command (indicated by the selector in the first argument) is applicable in the current context. For example, if the selector is `paste:` and there is no data in the pasteboard of a type the view can handle, the responder should return `NO` to suppress the Paste command. If the first responder does not implement the `canPerformAction:withSender:` method, or does not handle the given command, the message travels up the responder chain.

Listing 4-2 shows an implementation of the `canPerformAction:withSender:` method that looks for message matching the `cut:`, `copy:`, and `paste:` selectors; it enables or disables the Copy, Cut, and Paste menu commands based on the current selection context and, for paste, the contents of the pasteboard.

Listing 4-2 Conditionally enabling menu commands

```
- (BOOL)canPerformAction:(SEL)action withSender:(id)sender {
    BOOL retValue = NO;
    ColorTile *theTile = [self colorTileForOrigin:currentSelection];

    if (action == @selector(paste:) )
        retValue = (theTile == nil) &&
             [[UIPasteboard generalPasteboard] containsPasteboardTypes:
             [NSArray arrayWithObject:ColorTileUTI]];
    else if ( action == @selector(cut:) || action == @selector(copy:) )
        retValue = (theTile != nil);
    else
        retValue = [super canPerformAction:action withSender:sender];
    return retValue;
}
```

Note that the final `else` clause in this method calls the superclass implementation to give any superclasses a chance to handle commands that the subclass chooses to ignore.

Note that a menu command, when acted upon, can change the context for other menu commands. For example, if the user selects all objects in the view, the Copy and Cut commands should be included in the menu. In this case the responder can, while the menu is still visible, call `update` on the menu controller; this results in the reinvocation of `canPerformAction:withSender:` on the first responder.

Copying and Cutting the Selection

When users tap the Copy or Cut command of the editing menu, the system invokes the `copy:` or `cut:` method (respectively) of the responder object that implements it. Usually the first responder—your custom view—implements these methods, but if the first responder doesn't implement them, the message travels up the responder chain in the usual fashion. Note that the `UIResponderStandardEditActions`informal protocol declares these methods.

Note: Because `UIResponderStandardEditActions` is an informal protocol, any class in your application can implement its methods. But to take advantage of the default behavior for traversing the responder chain, the class implementing the methods should inherit from `UIResponder` and should be installed in the responder chain.

In response to a `copy:` or `cut:` message, you write the object or data represented by the selection to the pasteboard in as many different representations as you can. This operation involves the following steps (which assume a single pasteboard item):

1. From the selection, identify or obtain the object or the binary data corresponding to the object.

   Binary data must be encapsulated in an `NSData` object. If you're going to write another type of object to the pasteboard, it must be a property-list object—that is, an object of one of the following classes: `NSString`, `NSArray`, `NSDictionary`, `NSDate`, `NSNumber`, or `NSURL`. (For more on property-list objects, see *Property List Programming Guide*.)

2. If possible, generate one or more other representations of the object or data.

   For example, if in the previous step you created a `UIImage` object representing a selected image, you could use the `UIImageJPEGRepresentation` and `UIImagePNGRepresentation` functions to convert the image to a different representation.

3. Obtain a pasteboard object.

   In many cases, this is the general pasteboard, which you can get through the `generalPasteboard` class method.

4. Assign a suitable UTI for each representation of data written to the pasteboard item.

   See "Pasteboard Concepts" (page 24) for a discussion of this subject.

5. Write the data to the first pasteboard item for each representation type:

   - To write a data object, send a `setData:forPasteboardType:` message to the pasteboard object.
   - To write a property-list object, send a `setValue:forPasteboardType:` message to the pasteboard object.

6. If the command is Cut (`cut:` method), remove the object represented by the selection from the application's data model and update your view.

Listing 4-3 shows implementations of the `copy:` and `cut:` methods. The `cut:` method invokes the `copy:` method and then removes the selected object from the view and the data model. Note that the `copy:` method archives a custom object to obtain an `NSData` object that it can pass to the pasteboard in `setData:forPasteboardType:`.

Listing 4-3 Copying and cutting operations

```
- (void)copy:(id)sender {
    UIPasteboard *gpBoard = [UIPasteboard generalPasteboard];
    ColorTile *theTile = [self colorTileForOrigin:currentSelection];
    if (theTile) {
       NSData *tileData = [NSKeyedArchiver archivedDataWithRootObject:theTile];
        if (tileData)
```

```
            [gpBoard setData:tileData forPasteboardType:ColorTileUTI];
    }
}

- (void)cut:(id)sender {
    [self copy:sender];
    ColorTile *theTile = [self colorTileForOrigin:currentSelection];

    if (theTile) {
        CGPoint tilePoint = theTile.tileOrigin;
        [tiles removeObject:theTile];
         CGRect tileRect = [self rectFromOrigin:tilePoint inset:TILE_INSET];
        [self setNeedsDisplayInRect:tileRect];
    }
}
```

Pasting the Selection

When users tap the Paste command of the editing menu, the system invokes the `paste:` method of the responder object that implements it. Usually the first responder—your custom view—implements this method, but if the first responder doesn't implement it, the message travel up the responder in the usual fashion. The `paste:` method is declared by the `UIResponderStandardEditActions`informal protocol.

In response to a `paste:` message, you read an object from the pasteboard in a representation that your application supports. Then you add the pasted object to the application's data model and display the new object in the view in the user-indicated location. This operation involves the following steps (which assume a single pasteboard item):

1. Obtain a pasteboard object.

   In many cases, this is the general pasteboard, which you can get through the `generalPasteboard`class method.

2. Verify that the first pasteboard item contains data in a representation that your application can handle by calling the `containsPasteboardTypes:` method or the `pasteboardTypes` method and then examining the returned array of types.

   Note that you should have already performed this step in your implementation of `canPerformAction:withSender:`.

3. If the first item of the pasteboard contains data that the application can handle, call one of the following methods to read it:

   - `dataForPasteboardType:` if the data to be read is encapsulated in an `NSData`object.
   - `valueForPasteboardType:` if the data to be read is encapsulated in a property-list object (see "Copying and Cutting the Selection" (page 28)).

4. Add the object to the application's data model.

5. Display a representation of the object in the user interface at the location specified by the user.

Listing 4-4 is an example of an implementation of the `paste:` method. It does the reverse of the combined `cut:` and `copy:` methods. The custom view first sees whether the general pasteboard holds its custom representation of data; if it does, it then reads the data from the pasteboard, adds it to the application's data model, and marks part of itself—the current selection—for redrawing.

Listing 4-4 Pasting data to a selection

```
- (void)paste:(id)sender {
     UIPasteboard *gpBoard = [UIPasteboard generalPasteboard];
     NSArray *pbType = [NSArray arrayWithObject:ColorTileUTI];
     ColorTile *theTile = [self colorTileForOrigin:currentSelection];
     if (theTile == nil && [gpBoard containsPasteboardTypes:pbType]) {

        NSData *tileData = [gpBoard dataForPasteboardType:ColorTileUTI];
        ColorTile *theTile = (ColorTile *)[NSKeyedUnarchiver
unarchiveObjectWithData:tileData];
         if (theTile) {
             theTile.tileOrigin = self.currentSelection;
             [tiles addObject:theTile];
             CGRect tileRect = [self rectFromOrigin:currentSelection
inset:TILE_INSET];
             [self setNeedsDisplayInRect:tileRect];
         }
     }
}
```

Dismissing the Editing Menu

When your implementation of the `cut:`, `copy:` or `paste:` command returns, the editing menu is automatically hidden. You can keep it visible with the following line of code:

```
[UIMenuController setMenuController].menuVisible = YES;
```

The system may hide the editing menu at any time. For example, it hides the menu when an alert is displayed or the user taps in another area of the screen. If you have state or a display that depends on whether the editing menu is visible, you should listen for the notification named `UIMenuControllerWillHideMenuNotification` and take an appropriate action.

Adding Custom Edit Menu Items

You can add a custom item to the edit menu used for showing the system commands Copy, Cut, Paste, Select, Select All, and Delete. When users tap this item, a command is issued that affects the current target in an application-specific way. The UIKit framework accomplishes this through the target-action mechanism. The tap of an item results in a action message being sent to the first object in the responder chain that can handle the message. Figure 5-1 shows an example of a custom menu item ("Change Color").

Figure 5-1 An editing menu with a custom menu item

An instance of the `UIMenuItem` class represents a custom menu item. `UIMenuItem` objects have two properties, a title and an action selector, which you can change at any time. To implement a custom menu item, you must initialize a `UIMenuItem` instance with these properties, add the instance to the menu controller's array of custom menu items, and then implement the action method for handling the command in the appropriate responder subclass.

Other aspects of implementing a custom menu item are common to all code that uses the singleton`UIMenuController` object. In a custom or overridden view, you set the view to be the first responder, get the shared menu controller, set a target rectangle, and then display the editing menu with a call to `setMenuVisible:animated:`. The simple example in Listing 5-1 adds a custom menu item for changing a custom view's color between red and black.

Listing 5-1 Implementing a Change Color menu item

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *theTouch = [touches anyObject];
    if ([theTouch tapCount] == 2) {
        [self becomeFirstResponder];
       UIMenuItem *menuItem = [[UIMenuItem alloc] initWithTitle:@"Change Color"
 action:@selector(changeColor:)];
        UIMenuController *menuCont = [UIMenuController sharedMenuController];
        [menuCont setTargetRect:self.frame inView:self.superview];
        menuCont.arrowDirection = UIMenuControllerArrowLeft;
        menuCont.menuItems = [NSArray arrayWithObject:menuItem];
        [menuCont setMenuVisible:YES animated:YES];
    }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {}

- (BOOL)canBecomeFirstResponder { return YES; }
```

```
- (void)changeColor:(id)sender {
    if ([self.viewColor isEqual:[UIColor blackColor]]) {
        self.viewColor = [UIColor redColor];
    } else {
        self.viewColor = [UIColor blackColor];
    }
    [self setNeedsDisplay];
}
```

Note: The `arrowDirection` property of `UIMenuController`, shown in Listing 5-1, is new in iOS 3.2. It allows you to specify the direction the arrow attached to the editing menu points at its target rectangle. Also new is the Delete menu command; if users tap this menu command, the `delete:` method implemented by an object in the responder chain (if any) is invoked. The `delete:` method is declared in the `UIResponderStandardEditActions` informal protocol.

Using System Messaging Facilities

Sending a Mail Message

In iOS 3.0 and later, you can use the `MFMailComposeViewController` class to present a standard mail composition interface inside your own applications. Prior to displaying the interface, you use the methods of the class to configure the email recipients, the subject, body, and any attachments you want to include. Upon posting the interface (using the standard view controller techniques), the user has the option of editing the email contents before submitting the email to the Mail application for delivery. The user also has the option to cancel the email altogether.

Note: In all versions of iOS, you can also compose email messages by creating and opening a URL that uses the `mailto` scheme. URLs of this type are automatically handled by the Mail application. For more information on how to open a URL of this type, see "Communicating with Other Applications" in *iOS Application Programming Guide*.

To use the mail composition interface, you must add `MessageUI.framework` to your Xcode project and link against it in any relevant targets. To access the classes and headers of the framework, include an `#import <MessageUI/MessageUI.h>` statement at the top of any relevant source files. For information on how to add frameworks to your project, see "Files in Projects" in *Xcode Project Management Guide*.

To use the `MFMailComposeViewController` class in your application, you create an instance and use its methods to set the initial email data. You must also assign an object to the `mailComposeDelegate`property of the view controller to handle the dismissal of the interface when the user accepts or cancels the email. The delegate object you specify must conform to the `MFMailComposeViewControllerDelegate` protocol.

When specifying email addresses for the mail composition interface, you specify plain string objects. If you want to use email addresses from the user's list of contacts, you can use the Address Book framework to retrieve that information. For more information on how to get email and other information using this framework, see *Address Book Programming Guide for iOS*.

Listing 6-1 shows the code for creating the `MFMailComposeViewController` object and displaying the mail composition interface modally in your application. You would include the `displayComposerSheet` method in one of your custom view controllers and call the method as needed to display the interface. In this example, the parent view controller assigns itself as the delegate and implements the `mailComposeController:didFinishWithResult:error:` method. The delegate method dismisses the delegate without taking any further actions. In your own application, you could use the delegate to track whether the user sent or canceled the email by examining the value in the *`result`* parameter.

Listing 6-1 Posting the mail composition interface

```
@implementation WriteMyMailViewController (MailMethods)

-(void)displayComposerSheet
{
```

```
    MFMailComposeViewController *picker = [[MFMailComposeViewController alloc]
 init];
    picker.mailComposeDelegate = self;

    [picker setSubject:@"Hello from California!"];

    // Set up the recipients.
    NSArray *toRecipients = [NSArray arrayWithObjects:@"first@example.com",
                                    nil];
    NSArray *ccRecipients = [NSArray arrayWithObjects:@"second@example.com",
                                    @"third@example.com", nil];
    NSArray *bccRecipients = [NSArray arrayWithObjects:@"four@example.com",
                                    nil];

    [picker setToRecipients:toRecipients];
    [picker setCcRecipients:ccRecipients];
    [picker setBccRecipients:bccRecipients];

    // Attach an image to the email.
    NSString *path = [[NSBundle mainBundle] pathForResource:@"ipodnano"
                                   ofType:@"png"];
    NSData *myData = [NSData dataWithContentsOfFile:path];
    [picker addAttachmentData:myData mimeType:@"image/png"
                                   fileName:@"ipodnano"];

    // Fill out the email body text.
    NSString *emailBody = @"It is raining in sunny California!";
    [picker setMessageBody:emailBody isHTML:NO];

    // Present the mail composition interface.
    [self presentModalViewController:picker animated:YES];
    [picker release]; // Can safely release the controller now.
}

// The mail compose view controller delegate method
- (void)mailComposeController:(MFMailComposeViewController *)controller
              didFinishWithResult:(MFMailComposeResult)result
              error:(NSError *)error
{
    [self dismissModalViewControllerAnimated:YES];
}
@end
```

For more information on the standard view controller techniques for displaying interfaces, see *View Controller Programming Guide for iOS*. For information about the classes of the Message UI framework, see *Message UI Framework Reference*.

Sending an SMS Message

In iOS 4.0 and later, you can send text messages from within your application. This feature is strictly for sending messages. Incoming SMS messages go to the built-in Messages app.

To provide the standard user interface for composing an SMS (Short Message Service) message, use the `MFMessageComposeViewController` class in the Message UI framework. Create an instance of this class and assign it a delegate object. The delegate must conform to the `MFMessageComposeViewControllerDelegate` protocol.

Before presenting the composition interface to the user, you can configure initial recipients and message content. With setup complete, call the `UIViewController` `presentModalViewController:animated:` method to present the message composition view controller modally.

While the interface is visible, the user can edit the recipients list and message content. The user then sends the message, or cancels it, by tapping the appropriate control in the interface.

If the user requests that the message be sent, the system queues it for delivery and invokes the delegate object's `messageComposeViewController:didFinishWithResult:` method. The result is one of "sent," "cancelled," or "failed."

Finally, the delegate is responsible for dismissing the message composition view controller, which it should do by calling the `UIViewController` `dismissModalViewControllerAnimated:` method.

For a complete description of the `MFMessageComposeViewController` class, see *MFMessageComposeViewController Class Reference*.

Accessing Cellular Telephone Information

Use the Core Telephony framework to obtain information about a user's home cellular service provider—that is, the provider with whom the user has an account, as recorded on the device's SIM card. Providers can use this information to write applications that include services only for their own subscribers.

Use the `CTCarrier` class to obtain information from the installed SIM card about the user's cellular service provider, such as the provider name and whether it allows use of VoIP (Voice over Internet Protocol) on its network.

You can also use this framework to obtain information about the status of current cellular calls. If you are writing a VoIP (Voice over Internet Protocol) application, you may want to make use of this information. Use the `CTCallCenter` and `CTCall` classes to obtain information about current calls, including a unique identifier and state information for each call—dialing, incoming, connected, or disconnected.

For complete descriptions of the classes in the Core Telephony framework, see *Core Telephony Framework Reference*.

Document Revision History

This table describes the changes to *Device Features Programming Guide*.

Date	Notes
2010-04-30	New document that explains how to integrate device features into your application.

Document Revision History