External Accessory Programming Topics

Data Management: Device Information

2010-05-26

Contents

Listings

About External Accessories

The External Accessory framework (`ExternalAccessory.framework`) provides a conduit for communicating with accessories attached to any iOS-based device. Application developers can use this conduit to integrate accessory-level features into their applications.

Communicating with an external accessory requires you to work closely with the accessory manufacturer to understand the services provided by that accessory. Manufacturers must build explicit support into their accessory hardware for communicating with iOS. As part of this support, an accessory must support at least one command **protocol**, which is a custom scheme for sending data back and forth between the accessory and an attached application. Apple does not maintain a registry of protocols; it is up to the manufacturer to decide which protocols to support and whether to use custom protocols or standard protocols supported by other manufacturers.

As part of your communication with the accessory manufacturer, you must find out what protocols a given accessory supports. To prevent namespace conflicts, protocol names are specified as reverse-DNS strings of the form `com.apple.myProtocol`. This allows each manufacturer to define as many protocols as needed to support their line of accessories.

Note: If you are interested in becoming a developer of accessories for iPad, iPhone, or iPod touch, you can find information about how to do so on http://developer.apple.com.

Steps for Communicating with Accessories

An application communicates with an accessory by creating an `EASession` object for managing the accessory interactions. Session objects work with the underlying system to transfer data packets to and from the accessory. Data transfer in your application occurs through `NSInputStream` and `NSOutputStream` objects, which are vended by the session object once the connection is made. To receive data from the accessory, monitor the input stream using a custom delegate object. To send data to the accessory, write data packets to the output stream. The format of the incoming and outgoing data packets is determined by the protocol you use to communicate with the accessory.

For details on how to get a list of connected accessories and start a session with one of them, see "Connecting to an Accessory" (page 9).

Including the External Accessory Framework in Your Project

To use the features of the External Accessory framework, you must add `ExternalAccessory.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 <ExternalAccessory/ExternalAccessory.h>` statement at the top of any relevant source files. For more information on how to add frameworks to your project, see “Files in Projects” in *Xcode Project Management Guide*. For general information about the classes of the External Accessory framework, see *External Accessory Framework Reference*.

Declaring the Protocols Your Application Supports

Applications that are able to communicate with an external accessory should declare the protocols they support in their `Info.plist` file. Declaring support for specific protocols lets the system know that your application can be launched when that accessory is connected. If no application supports the connected accessory, the system may choose to launch the App Store and point out applications that do.

To declare the protocols your application supports, you must include the `UISupportedExternalAccessoryProtocols` key in your application’s `Info.plist` file. This key contains an array of strings that identify the communications protocols that your application supports. Your application can include any number of protocols in this list and the protocols can be in any order. The system does not use this list to determine which protocol your application should choose; it uses it only to determine if your application is capable of communicating with the accessory. It is up to your code to choose an appropriate communications protocol when it begins talking to the accessory.

For more information about the keys you put into your application’s `Info.plist` file, see *Information Property List Key Reference*.

Organization of This Document

This document includes the following articles:

- “Connecting to an Accessory” (page 9) describes the steps you use to connect to an accessory at runtime.
- “Monitoring Accessory-Related Events” (page 11) explains how to detect when accessories are connected or disconnected.

Connecting to an Accessory

Accessories are not visible through the External Accessory framework until they have been connected by the system and made ready for use. When an accessory does become visible, your application can get the appropriate accessory object and open a session using one or more of the protocols supported by the accessory.

The shared `EAAccessoryManager` object provides the main entry point for applications looking to communicate with accessories. This class contains an array of already connected accessory objects that you can enumerate to see if there is one your application supports. Most of the information in an `EAAccessory` object (such as the name, manufacturer, and model information) is intended for display purposes only. To determine whether your application can connect to an accessory, you must look at the accessory's protocols and see if there is one your application supports.

Note: It is possible for more than one accessory object to support the same protocol. If that happens, your code is responsible for choosing which accessory object to use.

For a given accessory object, only one session at a time is allowed for a specific protocol. The `protocolStrings` property of each `EAAccessory` object contains a dictionary whose keys are the supported protocols. If you attempt to create a session using a protocol that is already in use, the External Accessory framework generates an error.

Listing 1 shows a method that checks the list of connected accessories and grabs the first one that the application supports. It creates a session for the designated protocol and configures the input and output streams of the session. By the time this method returns the session object, it is connected to the accessory and ready to begin sending and receiving data.

Listing 1 Creating a communications session for an accessory

```
- (EASession *)openSessionForProtocol:(NSString *)protocolString
{
    NSArray *accessories = [[EAAccessoryManager sharedAccessoryManager]
                               connectedAccessories];
    EAAccessory *accessory = nil;
    EASession *session = nil;

    for (EAAccessory *obj in accessories)
    {
        if ([[obj protocolStrings] containsObject:protocolString])
        {
            accessory = obj;
            break;
        }
    }

    if (accessory)
    {
        session = [[EASession alloc] initWithAccessory:accessory
                                 forProtocol:protocolString];
        if (session)
        {
            [[session inputStream] setDelegate:self];
           [[session inputStream] scheduleInRunLoop:[NSRunLoop currentRunLoop]
                                  forMode:NSDefaultRunLoopMode];
            [[session inputStream] open];
            [[session outputStream] setDelegate:self];
          [[session outputStream] scheduleInRunLoop:[NSRunLoop currentRunLoop]
                                  forMode:NSDefaultRunLoopMode];
            [[session outputStream] open];
            [session autorelease];
        }
    }

    return session;
}
```

After the input and output streams are configured, the final step is to process the stream-related data. Listing 2 shows the fundamental structure of a delegate's stream processing code. This method responds to events from both input and output streams of the accessory. As the accessory sends data to your application an event arrives indicating there are bytes available to be read. Similarly, when the accessory is ready to receive data from your application, events arrive indicating that fact. (Of course, your application does not always have to wait for an event to arrive before it can write bytes to the stream. It can also call the stream's `hasBytesAvailable` method to see if the accessory is still able to receive data.) For more information on streams and handling stream-related events, see *Stream Programming Guide for Cocoa*.

Listing 2 Processing stream events

```
// Handle communications from the streams.
- (void)stream:(NSStream*)theStream handleEvent:(NSStreamEvent)streamEvent
{
    switch (streamEvent)
    {
        case NSStreamHasBytesAvailable:
            // Process the incoming stream data.
            break;

        case NSStreamEventHasSpaceAvailable:
            // Send the next queued command.
            break;

        default:
            break;
    }

}
```

Monitoring Accessory-Related Events

The External Accessory framework is capable of sending notifications whenever a hardware accessory is connected or disconnected. Although it is capable, it does not do so automatically. Your application must specifically request that notifications be generated by calling the `registerForLocalNotifications` method of the `EAAccessoryManager` class. When an accessory is connected, authenticated, and ready to interact with your application, the framework sends an `EAAccessoryDidConnectNotification` notification. When an accessory is disconnected, it sends an `EAAccessoryDidDisconnectNotification` notification. You can register to receive these notifications using the default `NSNotificationCenter`, and both notifications include information about which accessory was affected.

In addition to receiving notifications through the default notification center, an application that is currently interacting with an accessory can assign a delegate to the corresponding `EAAccessory` object and be notified of changes. Delegate objects must conform to the `EAAccessoryDelegate`protocol, which currently contains the optional `accessoryDidDisconnect:` method. You can use this method to receive disconnection notices without first setting up a notification observer.

If your application is suspended in the background when an accessory notification arrives, that notification is put in a queue. When your application begins running again (either in the foreground or background), notifications in the queue are delivered to your application. Notifications are also coalesced and filtered wherever possible to eliminate any irrelevant events. For example, if an accessory was connected and subsequently disconnected while your application was suspended, your application would ultimately not receive any indication that such events took place.

For more information about how to register to receive notifications, see *Notification Programming Topics*.

Document Revision History

This table describes the changes to *External Accessory Programming Topics*.

Date	**Notes**
2010-05-26	New document describing how to attach to external hardware devices.