Address Book Programming Guide for iOS

Data Management: Contact Data

2010-12-22

Contents

Figures

Introduction

The Address Book technology for iOS provides a way to store people's contact information and other personal information in a centralized database, and to share this information between applications. The technology has several parts:

- The Address Book framework provides access to the contact information.
- The Address Book UI framework provides the user interface to display the information.
- The Address Book database stores the information.
- The Contacts application provides a way for users to access their contact information.

This document covers the key concepts of the Address Book technology and explains the basic operations you can perform. When you add this technology to your application, users will be able to use the contact information that they use in other applications, such as Mail and Text, in your application. This document tells you how to do the following:

- Access the user's Address Book database
- Prompt the user for contact information
- Display contact information to the user
- Make changes to the user's Address Book database

To get the most out of this document, you should already understand navigation controllers and view controllers, and understand delegation and protocols.

Note: Developers who have used the Address Book technology on Mac OS X should be aware that the programming interface for this technology is different on iOS.

Organization of This Document

This document contains the following chapters:

- "Quick Start Tutorial" (page 9) gets you up and running by showing you how to create a simple application that uses the Address Book technology.
- "Building Blocks: Working with Records and Properties" (page 15) describes how to create an address book object, how to create person and group records, and how to get and set properties.

- "User Interaction: Prompting for and Displaying Data" (page 21) describes how to use the views provided by the Address Book UI framework to display a contact, let the user select a contact, create a new contact, and edit a contact.
- "Direct Interaction: Programmatically Accessing the Database" (page 27) describes the ways your application can read and write contact information directly.

See Also

The following documents discuss some of the fundamental concepts you should understand in order to get the most out of this document:

- *iOS Application Programming Guide* guides developers who are new to the iOS platform through the available technologies and how to use them to build applications. It includes relevant discussion of windows, views, and view controllers.
- *Interface Builder User Guide* explains how to use Interface Builder to create applications. It includes relevant discussion of the user interface for an application and making connections from the interface to the code.
- *Cocoa Fundamentals Guide* and *The Objective-C Programming Language* discuss many basic concepts you will need to write any application. It includes relevant discussion of delegation and protocols.

The following documents contain additional information about the Address Book frameworks:

- *Address Book Framework Reference for iOS* describes the API for direct interaction with records in the Address Book database.
- *Address Book UI Framework Reference for iOS* describes the controllers that facilitate displaying, editing, selecting, and creating records in the Address Book database, and their delegate protocols.

Quick Start Tutorial

In this tutorial, you will build a simple application that prompts the user to choose a person from his or her contacts list and then shows the chosen person's first and last name.

Create the Project

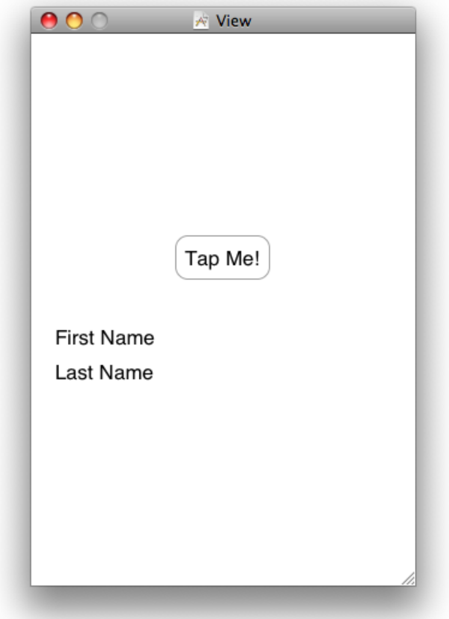
1. In Xcode, create a new project from the View Based Application template. Save the project as QuickStart.
2. The next step is to add the frameworks you will need. First, go to your project window and find the target named `QuickStart` in the Targets group. Open its info panel (File > Get Info) and, in the General tab, you will see a list of linked frameworks.
3. Add the Address Book and Address Book UI frameworks to the list of linked frameworks, by clicking the plus button and selecting them from the list.

Important: The project will fail to build (with a linker error) if the Address Book and Address Book UI frameworks are not in this list of frameworks to link against.

Lay Out the View

1. Open the nib file named `QuickStartViewController.xib` in Interface Builder.
2. Add a button and two text labels to the view by dragging them in from the library panel. Then arrange them as shown in Figure 1-1.
3. Save the nib file and return to Xcode.

Figure 1-1 Laying out the view in Interface Builder



You have now created an interface that users can use to interact with your application. In the next section, you will write the code that runs underneath the interface. In the last section, you will complete this nib file by making the connections between the code and the interface.

Write the Header File

Add the following code to `QuickStartViewController.h`, the header file for the view controller. This code declares the outlets for the labels and the action for the button that you just created in Interface Builder. It also declares that this view controller adopts the `ABPeoplePickerNavigationControllerDelegate` protocol, which you will implement next.

```
#import <UIKit/UIKit.h>
#import <AddressBook/AddressBook.h>
#import <AddressBookUI/AddressBookUI.h>


@interface QuickStartViewController : UIViewController
<ABPeoplePickerNavigationControllerDelegate> {
    IBOutlet UILabel *firstName;
    IBOutlet UILabel *lastName;
}

@property (nonatomic, retain) UILabel *firstName;
@property (nonatomic, retain) UILabel *lastName;

- (IBAction)showPicker:(id)sender;

@end
```

You can use the header file for the application delegate just as the template made it.

Write the Implementation File

Add the following code to `QuickStartViewController.m`, the implementation file for the view controller. This code synthesizes the accessor methods for `firstName` and `lastName` and implements the `showPicker:` method, which is called when the user taps the Tap Me! button. The `showPicker:` method creates a new people picker, sets the view controller as its delegate, and presents the picker as a modal view controller.

```
#import "QuickStartViewController.h"

@implementation QuickStartViewController

@synthesize firstName;
@synthesize lastName;


- (IBAction)showPicker:(id)sender {
    ABPeoplePickerNavigationController *picker =
            [[ABPeoplePickerNavigationController alloc] init];
    picker.peoplePickerDelegate = self;

    [self presentModalViewController:picker animated:YES];
    [picker release];
}
```

Continuing to add code to the same file, you will now begin implementing the delegate protocol, by adding two more methods. If the user cancels, the first method is called to dismiss the people picker. If the user selects a person, the second method is called to copy the first and last name of the person into the labels and dismiss the people picker.

Note: This function `ABRecordCopyValue` is used to get any property from a person record or a group record; it is used here to show the most general case. However, in actual applications, the function `ABRecordCopyCompositeName` is the recommended way to get a person's full name to display. It puts the first and last name in the order preferred by the user, which provides a more uniform user experience.

```
- (void)peoplePickerNavigationControllerDidCancel:
            (ABPeoplePickerNavigationController *)peoplePicker {
    [self dismissModalViewControllerAnimated:YES];
}


- (BOOL)peoplePickerNavigationController:
            (ABPeoplePickerNavigationController *)peoplePicker
      shouldContinueAfterSelectingPerson:(ABRecordRef)person {

    NSString* name = (NSString *)ABRecordCopyValue(person,
                                         kABPersonFirstNameProperty);
    self.firstName.text = name;
    [name release];

    name = (NSString *)ABRecordCopyValue(person, kABPersonLastNameProperty);
    self.lastName.text = name;
    [name release];

    [self dismissModalViewControllerAnimated:YES];

    return NO;
}
```

To fully implement the delegate protocol, you must also add one more method. The people picker calls this method when the user taps on a property of the selected person in the picker. In this application, the people picker is always dismissed when the user selects a person, so there is no way for the user to select a property of that person. This means that the method will never be called in this particular application. However if it were left out, the implementation of the protocol would be incomplete.

```
- (BOOL)peoplePickerNavigationController:
            (ABPeoplePickerNavigationController *)peoplePicker
      shouldContinueAfterSelectingPerson:(ABRecordRef)person
                                property:(ABPropertyID)property
                              identifier:(ABMultiValueIdentifier)identifier{
    return NO;
}
```

Finally, release allocated memory.

```
- (void)dealloc {
    [firstName release];
    [lastName release];
    [super dealloc];
}

@end
```

You can use the implementation file for the application delegate just as the template made it.

Make Connections in Interface Builder

In "Lay Out the View" (page 9), you created an interface for your application. In "Write the Header File" (page 10) and "Write the Implementation File" (page 11), you wrote the code to drive that interface. Now you will complete the application by making the connections between the interface and the code.

1. Open the nib file named `QuickStartViewController.xib` in Interface Builder.
2. In the Identity inspector (Tools > Identity Inspector), verify that the class identity of File's Owner is `QuickStartViewController`—it should already be set correctly for you by the template.
3. Control-click (or right-click) on File's Owner and connect the outlets for `firstName` and `lastName` from File's Owner to the first name and last name labels.
4. Control-click on the "Tap Me!" button and connect the Touch Up Inside outlet from the button to File's Owner, selecting `showPicker` as its action.
5. Save the nib file.

For information about outlets and making connections between them, see "Creating and Managing Outlet and Action Connections" in *Interface Builder User Guide*.

Build and Run the Application

When you run the application, the first thing you see is a button and two empty text labels. Tapping the button brings up the people picker. When you select a person, the people picker goes away and the first and last name of the person you selected are displayed in the labels.

In this chapter, you learned to perform a fairly simple task using the two Address Book frameworks. You used the default Xcode template, and added code to present a people picker and adopt the `ABPeoplePickerNavigationControllerDelegate` protocol. The concepts used in this simple example can easily be extended for a variety of other uses by your own applications.

Building Blocks: Working with Records and Properties

There are four basic kinds of objects that you need to understand in order to interact fully with the Address Book database: address books, records, single-value properties, and multivalue properties. This chapter discusses how data is stored in these objects and describes the functions used to interact with them.

For information on how to interact directly with the Address Book database (for example to add or remove person records), see "Direct Interaction: Programmatically Accessing the Database" (page 27).

Address Books

Address books objects let you interact with the Address Book database. To use an address book, declare an instance of `ABAddressBookRef` and set it to the value returned from the function `ABAddressBookCreate`. You can create multiple address book objects, but they are all backed by the same shared database.

Important: Instances of `ABAddressBookRef` cannot be used by multiple threads. Each thread must make its own instance.

After you have created an address book reference, your application can read data from it and save changes to it. To save the changes, use the function `ABAddressBookSave`; to abandon them, use the function `ABAddressBookRevert`. To check whether there are unsaved changes, use the function `ABAddressBookHasUnsavedChanges`.

The following code listing illustrates a common coding pattern for making and saving changes to the address book database:

```
ABAddressBookRef addressBook;
bool wantToSaveChanges = YES;
bool didSave;
CFErrorRef error = NULL;

addressBook = ABAddressBookCreate();

/* ... Work with the address book. ... */

if (ABAddressBookHasUnsavedChanges(addressBook)) {
    if (wantToSaveChanges) {
        didSave = ABAddressBookSave(addressBook, &error);
        if (!didSave) {/* Handle error here. */}
    } else {
        ABAddressBookRevert(addressBook);
    }
}

CFRelease(addressBook);
```

Your application can request to receive a notification when another application (or another thread in the same application) makes changes to the Address Book database. In general, you should register for a notification if you are displaying existing contacts and you want to update the UI to reflect changes to the contacts that may happen while your application is running.

Use the function `ABAddressBookRegisterExternalChangeCallback` to register a function of the prototype `ABExternalChangeCallback`. You may register multiple change callbacks by calling `ABAddressBookRegisterExternalChangeCallback` multiple times with different callbacks or contexts. You can also unregister the function using `ABAddressBookUnregisterExternalChangeCallback`.

When you receive a change callback, there are two things you can do: If you have no unsaved changes, your code should simply revert your address book to get the most up-to-date data. If you have unsaved changes, you may not want to revert and lose those changes. If this is the case you should save, and the Address Book database will do its best to merge your changes with the external changes. However, you should be prepared to take other appropriate action if the changes cannot be merged and the save fails.

Records

In the Address Book database, information is stored in records, represented by `ABRecordRef` objects. Each record represents a person or group. The function `ABRecordGetRecordType` returns `kABPersonType` if the record is a person, and `kABGroupType` if it is a group. Developers familiar with the Address Book technology on Mac OS should note that there are not separate classes for different types of records; both person objects and group objects are instances of the same class.

Important: Record objects cannot be passed across threads safely. Instead, you should pass the corresponding record identifier. See "Using Record Identifiers" (page 27) for more information.

Even though records are usually part of the Address Book database, they can also exist outside of it. This makes them a useful way to store contact information your application is working with.

Within a record, the data is stored as a collection of properties. The properties available for group and person objects are different, but the functions used to access them are the same. The functions `ABRecordCopyValue` and `ABRecordSetValue` get and set properties, respectively. Properties can also be removed completely, using the function `ABRecordRemoveValue`.

Person Records

Person records are made up of both single-value and multivalue properties. Properties that a person can have only one of, such as first name and last name, are stored as single-value properties. Other properties that a person can have more that one of, such as street address and phone number, are multivalue properties. The properties for person records are listed in several sections in "Constants" in *ABPerson Reference*.

For more information about functions related to directly editing the contents of person records, see "Working with Person Records" (page 27).

Group Records

Users may organize their contacts into groups for a variety of reasons. For example, a user may create a group containing coworkers involved in a project, or members of a sports team they play on. Your application can use groups to allow the user to perform an action for several contacts in their address book at the same time.

Group records have only one property, `kABGroupNameProperty`, which is the name of the group. To get all the people in a group, use the function `ABGroupCopyArrayOfAllMembers` or `ABGroupCopyArrayOfAllMembersWithSortOrdering`, which return a `CFArrayRef` of `ABRecordRef` objects.

For more information about functions related to directly editing the contents of group records, see "Working with Group Records" (page 28).

Properties

There are two basic types of properties, single-value and multivalue. Single-value properties contain data that can only have a single value, such as a person's name. Multivalue properties contain data that can have multiple values, such as a person's phone number. Multivalue properties can be either mutable or immutable.

For a list of the properties for person records, see many of the sections within "Constants" in *ABPerson Reference*. For properties of group records, see "Group Properties" in *ABGroup Reference*.

Single-Value Properties

The following code listing illustrates getting and setting the value of a single-value property:

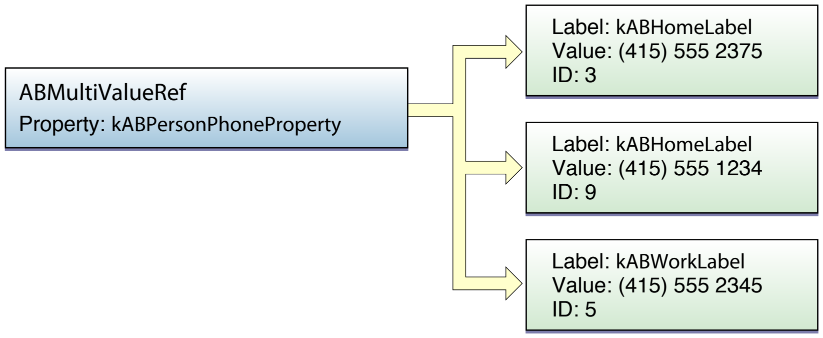
```
ABRecordRef aRecord = ABPersonCreate();
CFErrorRef anError = NULL;
bool didSet;

didSet = ABRecordSetValue(aRecord, kABPersonFirstNameProperty, CFSTR("Katie"),
 &anError);
if (!didSet) {/* Handle error here. */}

didSet = ABRecordSetValue(aRecord, kABPersonLastNameProperty, CFSTR("Bell"),
&anError);
if (!didSet) {/* Handle error here. */}

CFStringRef firstName, lastName;
firstName = ABRecordCopyValue(aRecord, kABPersonFirstNameProperty);
lastName  = ABRecordCopyValue(aRecord, kABPersonLastNameProperty);

/* ... Do something with firstName and lastName. ... */

CFRelease(aRecord);
CFRelease(firstName);
CFRelease(lastName);
```

Multivalue Properties

Multivalue properties consist of a list of values. Each value has a text label and an identifier associated with it. There can be more than one value with the same label, but the identifier is always unique. There are constants defined for some commonly used text labels—see "Generic Property Labels" in *ABPerson Reference*.

For example, Figure 2-1 shows a phone number property. Here, a person has multiple phone numbers, each of which has a text label, such as home or work, and an identifier. Note that there are two home phone numbers in this example; they have the same label but different identifiers.

Figure 2-1 Multivalue properties



The individual values of a multivalue property are referred to by identifier or by index, depending on the context. Use the functions `ABMultiValueGetIndexForIdentifier` and `ABMultiValueGetIdentifierAtIndex` to convert between indices and multivalue identifiers.

To keep a reference to a particular value in the multivalue property, store its identifier. The index will change if values are added or removed. The identifier is guaranteed not to change except across devices.

The following functions let you read the contents of an individual value, which you specify by its index:

- `ABMultiValueCopyLabelAtIndex` and `ABMultiValueCopyValueAtIndex` copy individual values.
- `ABMultiValueCopyArrayOfAllValues` copies all of the values into an array.

Mutable Multivalue Properties

Multivalue objects are immutable; to change one you need to make a mutable copy using the function `ABMultiValueCreateMutableCopy`. You can also create a new mutable multivalue object using the function `ABMultiValueCreateMutable`.

The following functions let you modify mutable multivalue properties:

- `ABMultiValueAddValueAndLabel` and `ABMultiValueInsertValueAndLabelAtIndex` add values.
- `ABMultiValueReplaceValueAtIndex` and `ABMultiValueReplaceLabelAtIndex` change values.
- `ABMultiValueRemoveValueAndLabelAtIndex` removes values.

The following code listing illustrates getting and setting a multivalue property:

```
ABMutableMultiValueRef multi =
        ABMultiValueCreateMutable(kABMultiStringPropertyType);
CFErrorRef anError = NULL;
ABMultiValueIdentifier multivalueIdentifier;
bool didAdd, didSet;

// Here, multivalueIdentifier is just for illustration purposes; it isn't
// used later in the listing.  Real-world code can use this identifier to
// reference the newly-added value.
didAdd = ABMultiValueAddValueAndLabel(multi, @"(555) 555-1234",
                      kABPersonPhoneMobileLabel, &multivalueIdentifier);
if (!didAdd) {/* Handle error here. */}

didAdd = ABMultiValueAddValueAndLabel(multi, @"(555) 555-2345",
                      kABPersonPhoneMainLabel, &multivalueIdentifier);
if (!didAdd) {/* Handle error here. */}

ABRecordRef aRecord = ABPersonCreate();
didSet = ABRecordSetValue(aRecord, kABPersonPhoneProperty, multi, &anError);
if (!didSet) {/* Handle error here. */}
CFRelease(multi);

/* ... */

CFStringRef phoneNumber, phoneNumberLabel;
multi = ABRecordCopyValue(aRecord, kABPersonPhoneProperty);

for (CFIndex i = 0; i < ABMultiValueGetCount(multi); i++) {
    phoneNumberLabel = ABMultiValueCopyLabelAtIndex(multi, i);
    phoneNumber      = ABMultiValueCopyValueAtIndex(multi, i);

    /* ... Do something with phoneNumberLabel and phoneNumber. ... */

    CFRelease(phoneNumberLabel);
    CFRelease(phoneNumber);
}

CFRelease(aRecord);
CFRelease(multi);
```

Street Addresses

Street addresses are represented as a multivalue of dictionaries. All of the above discussion of multivalues still applies to street addresses. Each of the values has a label, such as home or work (see “Generic Property Labels” in *ABPerson Reference*), and each value in the multivalue is a street address stored as a dictionary. Within the value, the dictionary contains keys for the different parts of a street address, which are listed in “Address Property” in *ABPerson Reference*.

The following code listing shows how to set and display a street address:

```
ABMutableMultiValueRef address =
        ABMultiValueCreateMutable(kABDictionaryPropertyType);

// Set up keys and values for the dictionary.
CFStringRef keys[5];
CFStringRef values[5];
keys[0] = kABPersonAddressStreetKey;
```

```
keys[1] = kABPersonAddressCityKey;
keys[2] = kABPersonAddressStateKey;
keys[3] = kABPersonAddressZIPKey;
keys[4] = kABPersonAddressCountryKey;
values[0] = CFSTR("1234 Laurel Street");
values[1] = CFSTR("Atlanta");
values[2] = CFSTR("GA");
values[3] = CFSTR("30303");
values[4] = CFSTR("USA");

CFDictionaryRef aDict = CFDictionaryCreate(
        kCFAllocatorDefault,
        (void *)keys,
        (void *)values,
        5,
        &kCFCopyStringDictionaryKeyCallBacks,
        &kCFTypeDictionaryValueCallBacks
);

// Add the street address to the multivalue.
ABMultiValueIdentifier identifier;
bool didAdd;
didAdd = ABMultiValueAddValueAndLabel(address, aDict, kABHomeLabel, &identifier);
if (!didAdd) {/* Handle error here. */}
CFRelease(aDict);

/* ... Do something with the multivalue, such as adding it to a person record.
 ...*/

CFRelease(address);
```

User Interaction: Prompting for and Displaying Data

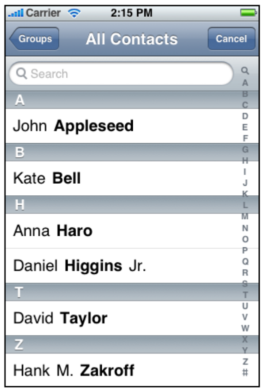
The Address Book UI framework provides three view controllers and one navigation controller for common tasks related to working with the Address Book database and contact information. By using these controllers rather than creating your own, you reduce the amount of work you have to do and provide your users with a more consistent experience.

This chapter includes some short code listings you can use as a starting point. For a fully worked example, see *QuickContacts*.

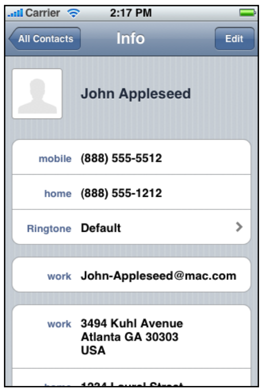
What's Available

The Address Book UI framework provides four controllers:

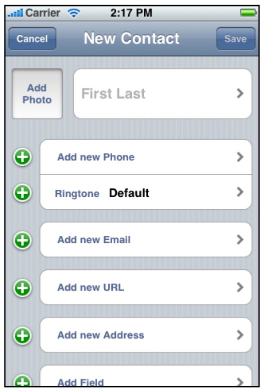
- `ABPeoplePickerNavigationController` prompts the user to select a person record from their address book.
- `ABPersonViewController` displays a person record to the user and optionally allows editing.
- `ABNewPersonViewController` prompts the user create a new person record.
- `ABUnknownPersonViewController` prompts the user to complete a partial person record, optionally allows them to add it to the address book.



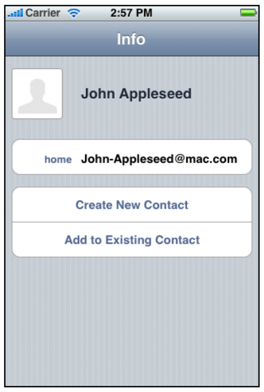
People picker



Person View Controller



New-Person View Controller



Unknown-Person View Controller

To use these controllers, you must set a delegate for them which implements the appropriate delegate protocol. You should not need to subclass these controllers; the expected way to modify their behavior is by your implementation of their delegate. In this chapter, you will learn more about these controllers and how to use them.

For more information about delegation, see "Delegates and Data Sources" in *Cocoa Fundamentals Guide*. For more information about protocols, see Protocols in *The Objective-C Programming Language*.

Prompting the User to Choose a Person Record

The `ABPeoplePickerNavigationController` class allows users to browse their list of contacts and select a person and, at your option, one of that person's properties. To use a people picker, do the following:

1. Create and initialize an instance of the class.

2. Set the delegate, which must adopt the `ABPeoplePickerNavigationControllerDelegate` protocol.

3. Optionally, set `displayedProperties` to the array of properties you want displayed. The relevant constants are defined as integers; wrap them in an `NSNumber` object using the `numberWithInt:` method to get an object that can be put in an array.

4. Present the people picker as a modal view controller using the `presentModalViewController:animated:` method. It is recommended that you present it using animation.

The following code listing shows how a view controller which implements the `ABPeoplePickerNavigationControllerDelegate` protocol can present a people picker:

```
ABPeoplePickerNavigationController *picker =
        [[ABPeoplePickerNavigationController alloc] init];
picker.peoplePickerDelegate = self;
[self presentModalViewController:picker animated:YES];
[picker release];
```

The people picker calls one of its delegate's methods depending on the user's action:

- If the user cancels, the people picker calls the method `peoplePickerNavigationControllerDidCancel:` of the delegate, which should dismiss the people picker.

- If the user selects a person, the people picker calls the method `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:` of the delegate to determine if the people picker should continue. To prompt the user to choose a specific property of the selected person, return `YES`. Otherwise return `NO` and dismiss the picker.

- If the user selects a property, the people picker calls the method `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:property:identifier:` of the delegate to determine if it should continue. To perform the default action (dialing a phone number, starting a new email, etc.) for the selected property, return `YES`. Otherwise return `NO` and dismiss the picker using the `dismissModalViewControllerAnimated:` method. It is recommended that you dismiss it using animation..

Displaying and Editing a Person Record

The `ABPersonViewController` class displays a record to the user. To use this controller, do the following:

1. Create and initialize an instance of the class.
2. Set the delegate, which must adopt the `ABPersonViewControllerDelegate` protocol. To allow the user to edit the record, set `allowsEditing` to `YES`.
3. Set the `displayedPerson` property to the person record you want to display.
4. Optionally, set `displayedProperties` to the array of properties you want displayed. The relevant constants are defined as integers; wrap them in an `NSNumber` object using the `numberWithInt:` method to get an object that can be put in an array.
5. Display the person view controller using the `pushViewController:animated:` method of the current navigation controller. It is recommended that you present it using animation.

Important: Person view controllers must be used with a navigation controller in order to function properly.

The following code listing shows how a navigation controller can present a person view controller:

```
ABPersonViewController *view = [[ABPersonViewController alloc] init];

view.personViewDelegate = self;
view.displayedPerson = person; // Assume person is already defined.

[self.navigationController pushViewController:view animated:YES];
[view release];
```

If the user taps on a property in the view, the person view controller calls the `personViewController:shouldPerformDefaultActionForPerson:property:identifier:` method of the delegate to determine if the default action for that property should be taken. To perform the default action for the selected property, such as dialing a phone number or composing a new email, return `YES`; otherwise return `NO`.

Prompting the User to Create a New Person Record

The `ABNewPersonViewController` class allows users to create a new person. To use it, do the following:

1. Create and initialize an instance of the class.
2. Set the delegate, which must adopt the `ABNewPersonViewControllerDelegate` protocol. To populate fields, set the value of `displayedPerson`. To put the new person in a particular group, set `parentGroup`.
3. Create and initialize a new navigation controller, and set its root view controller to the new-person view controller

4. Present the navigation controller as a modal view controller using the `presentModalViewController:animated:` method. It is recommended that you present it using animation.

Important: New-person view controllers must be used with a navigation controller in order to function properly. It is recommended that you present a new-person view controller modally.

The following code listing shows how a navigation controller can present a new person view controller:

```
ABNewPersonViewController *view = [[ABNewPersonViewController alloc] init];
view.newPersonViewDelegate = self;

UINavigationController *newNavigationController = [[UINavigationController alloc]

initWithRootViewController:view];
[self presentModalViewController:newNavigationController
                        animated:YES];

[view release];
[newNavigationController release];
```

When the user taps the Save or Cancel button, the new-person view controller calls the method `newPersonViewController:didCompleteWithNewPerson:` of the delegate, with the resulting person record. If the user saved, the new record is first added to the address book. If the user cancelled, the value of `person` is `NULL`. The delegate must dismiss the new-person view controller using the navigation controller's `dismissModalViewControllerAnimated:` method. It is recommended that you dismiss it using animation.

Prompting the User to Create a New Person Record from Existing Data

The `ABUnknownPersonViewController` class allows the user to add data to an existing person record or to create a new person record for the data. To use it, do the following:

1. Create and initialize an instance of the class.
2. Create a new person record and populate the properties to be displayed.
3. Set `displayedPerson` to the new person record you created in the previous step.
4. Set the delegate, which must adopt the `ABUnknownPersonViewControllerDelegate` protocol.
5. To allow the user to add the information displayed by the unknown-person view controller to an existing contact or to create a new contact with them, set `allowsAddingToAddressBook` to `YES`.
6. Display the unknown-person view controller using the `pushViewController:animated:` method of the navigation controller. It is recommended that you present it using animation.

Important: Unknown-person view controllers must be used with a navigation controller in order to function properly.

The following code listing shows how you can present an unknown-person view controller:

```
ABUnknownPersonViewController *view = [[ABUnknownPersonViewController alloc]
init];

view.unknownPersonViewDelegate = self;
view.displayedPerson = person; // Assume person is already defined.
view.allowsAddingToAddressBook = YES;

[self.navigationController pushViewController:view animated:YES];
[view release];
```

When the user finishes creating a new contact or adding the properties to an existing contact, the unknown-person view controller calls the method `unknownPersonViewController:didResolveToPerson:` of the delegate with the resulting person record. If the user canceled, the value of `person` is `NULL`.

Direct Interaction: Programmatically Accessing the Database

Although many common Address Book database tasks depend on user interaction, in some cases appropriate for the application needs to interact with the Address Book database directly. There are several functions in the Address Book framework that provide this ability.

In order to provide a uniform user experience, it is important to use these functions only when they are appropriate. Rather than using these functions to create new view or navigation controllers, your program should call the provided view or navigation controllers whenever possible. For more information, see "User Interaction: Prompting for and Displaying Data" (page 21).

Remember that the Address Book database is ultimately owned by the user, so applications must be careful not to make unexpected changes to it. Generally, changes should be initiated or confirmed by the user. This is especially true for groups, because there is no interface on the device for the user to manage groups and undo your application's changes.

Using Record Identifiers

Every record in the Address Book database has a unique record identifier. This identifier always refers to the same record, unless that record is deleted or the MobileMe sync data is reset. Record identifiers can be safely passed between threads. They are not guaranteed to remain the same across devices.

The recommended way to keep a long-term reference to a particular record is to store the first and last name, or a hash of the first and last name, in addition to the identifier. When you look up a record by ID, compare the record's name to your stored name. If they don't match, use the stored name to find the record, and store the new ID for the record.

To get the record identifier of a record, use the function `ABRecordGetRecordID`. To find a person record by identifier, use the function `ABAddressBookGetPersonWithRecordID`. To find a group by identifier, use the function `ABAddressBookGetGroupWithRecordID`. To find a person record by name, use the function `ABAddressBookCopyPeopleWithName`.

Working with Person Records

You can add and remove records from the Address Book database using the functions `ABAddressBookAddRecord` and `ABAddressBookRemoveRecord`.

There are two ways to find a person record in the Address Book database: by name, using the function `ABAddressBookCopyPeopleWithName`, and by record identifier, using the function `ABAddressBookGetPersonWithRecordID`. To accomplish other kinds of searches, use the function `ABAddressBookCopyArrayOfAllPeople` and then filter the results using the `NSArray` method `filteredArrayUsingPredicate:`.

To sort an array of people, use the function `CFArraySortValues` with the function `ABPersonComparePeopleByName` as the comparator and a context of the type `ABPersonSortOrdering`. The user's desired sort order, as returned by `ABPersonGetSortOrdering`, is generally the preferred context.

The following code listing shows an example of sorting the entire Address Book database:

```
ABAddressBookRef addressBook = ABAddressBookCreate();
CFArrayRef people = ABAddressBookCopyArrayOfAllPeople(addressBook);
CFMutableArrayRef peopleMutable = CFArrayCreateMutableCopy(
                                          kCFAllocatorDefault,
                                          CFArrayGetCount(people),
                                          people
                                  );


CFArraySortValues(
        peopleMutable,
        CFRangeMake(0, CFArrayGetCount(peopleMutable)),
        (CFComparatorFunction) ABPersonComparePeopleByName,
        (void*) ABPersonGetSortOrdering()
);

CFRelease(addressBook);
CFRelease(people);
CFRelease(peopleMutable);
```

Working with Group Records

You can find a specific group by record identifier using the function `ABAddressBookGetGroupWithRecordID`. You can also retrieve an array of all the groups in an address book using `ABAddressBookCopyArrayOfAllGroups`, and get a count of how many groups there are in an address book using the function `ABAddressBookGetGroupCount`.

You can modify the members of a group programatically. To add a person to a group, use the function `ABGroupAddMember`; to remove a person from a group, use the function `ABGroupRemoveMember`. Before a person record can be added to a group, it must already be in the Address Book database. If you need to add a new person record to a group and to the database at the same time, you must first add it to the address book database, save the database, and then add the person record to the group.

Document Revision History

This table describes the changes to *Address Book Programming Guide for iOS*.

Date	Notes
2010-12-22	Corrected minor errors in code listings.
2010-11-15	Corrected minor error in code listing. Other minor changes throughout.
2010-07-08	Changed the title from "Address Book Programming Guide for iPhone OS."
2010-03-24	Added example code to the Interacting Using UI Controllers section.
2009-10-05	Minor changes to code listing.
2009-05-27	Added discussion about the return value of ABMultiValueCopyLabelAtIndex. Corrected notes about how to use AddressBookUI view controllers.
2009-05-06	Made minor corrections to discussion of record identifiers. Small wording changes for clarity throughout.
2009-02-04	Minor restructuring for better readability.
2008-10-15	Added example code for working with street addresses. Other minor changes throughout.
2008-09-09	Minor update for iOS 2.1.
2008-07-31	Minor wording changes. Corrected typos. Reordered content in "Working with Address Book Objects."
2008-07-08	Updated example code. Made small editorial and structural changes throughout.
2008-06-06	New document that explains how to work with Address Book records, and use views to display and prompt for contact information.