Address Book Programming Guide for iPhone OS

Data Management: Contact Data

2008-10-15

Contents

Figures

Introduction

The Address Book technology for iPhone OS provides a way to store contact information and other personal information about people 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 be familiar with the basics of development for the iPhone, understand navigation controllers and view controllers, and understand delegation and protocols.

Note: Developers familiar with the Address Book technology on Mac OS should be aware that although the Address Book technology stores the same data on both platforms, the API is different. If you have used this technology on Mac OS, you may already understand the basic concepts of how to interact with the Address Book database, but the specific API on iPhone OS will be new to you.

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.
- "Working with Address Book Objects" (page 13) describes how to create an address book object, how to create person and group objects, and how to get and set properties.
- "Interacting Using UI Controllers" (page 19) describes how to use the views provided in the Address Book UI framework to display a contact, let the user select a contact, create a new contact, and edit a contact.

- "Interacting Directly with the Address Book Database" (page 25) describes the ways your application can interact with person and group records 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:

- *iPhone Application Programming Guide* guides developers who are new to the iPhone OS platform through the available technologies and how to use them to build applications. It includes relevant discussion of the 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 2.0 Programming Language* discuss many basic concepts you will need to write any application. It includes relevant discussion of delegation and prococols.

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

- *Address Book Framework Reference* describes the API for direct interaction with records in the Address Book database.
- *Address Book UI Framework Reference* 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.

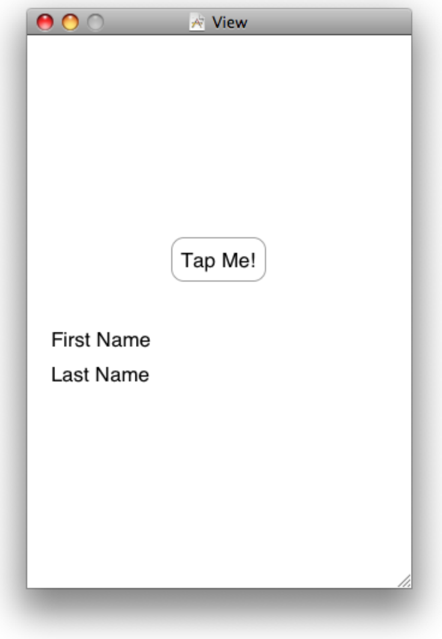
Create the Project

In Xcode, create a new project from the View Based Application template. Save the project as QuickStart. 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, there is a list of linked libraries. Add the Address Book and Address Book UI frameworks by clicking the plus button and selecting them from the list.

Lay Out the View

Next, lay out the interface for your project. In the Resources folder of the project window, open the Interface Builder nib file named `QuickStartViewController.xib`. Add a button and two text labels to the view by dragging them in from the library. Then arrange them as shown in Figure 1-1.

Figure 1-1 Laying out the view in Interface Builder



Now you have created an interface that you 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. For now, save the file and return to Xcode.

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

Next, 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 will be 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. The people picker will call the first function when the user cancels, which should dismiss the people picker. The people picker will call the second function when the user selects a person, which copies the first and last name of the person into the labels and dismisses the people picker.

Note: In this example, the function `ABRecordCopyValue` is used to get the name to illustrate the use of the general function. It is used to get any property from a person record or a group record. 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 following function. The people picker would call this third function when the user tapped on a property of the selected person in the picker. In this application, the people picker is 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 third method can never be called. However if it were left out, the implementation of the protocol would be incomplete, causing a compiler warning.

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

There are two additional functions from template; you can use them just as they are:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
            (UIInterfaceOrientation)interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}


- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; // Releases the view if it
                                     // doesn't have a superview

    // Release anything that's not essential, such as cached data
}
```

Finally, release everything so that you don't leak 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 10), you wrote the code to drive that interface. Now you will complete the application by returning to Interface Builder to make the connections between the interface and the code.

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. Connect the outlets for `firstName` and `lastName` from File's Owner to the first name and last name labels. Finally, connect the Touch Up Inside outlet from the button to File's Owner and select the `showPicker` method.

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

Build and Run the Application

When you run the application, the initial view you see is the one with 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.

This is conceptually a fairly simple task, and was fairly simple to do using the two Address Book frameworks. You had to add code to the template to present a people picker and adopt the `ABPeoplePickerNavigationControllerDelegate` protocol. This concepts used in this simple example can easily be extended for a variety of other uses by your application.

Working with Address Book Objects

There are four basic objects that you need to understand in order to interact fully with the Address Book database. This chapter discusses how data is stored in these objects, and 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 "Interacting Directly with the Address Book Database" (page 25).

Address books	Let you interact with the Address Book database and save changes to it. Made up of records.
Records	Represent a person or a group. Made up of single-value and multivalue properties.
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 numbers. Made up of values.

Address Books

To use an address book, declare an instance of `ABAddressBookRef` and set it to the value returned from the function `ABAddressBookCreate`.

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

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`.

Here is a code sample that illustrates a common coding pattern for making and saving changes to the address book database:

```
ABAddressBookRef addressBook;
CFErrorRef error = NULL;
BOOL wantToSaveChanges = YES;

addressBook = ABAddressBookCreate();

/* ... work with address book ... */

if (ABAddressBookHasUnsavedChanges(addressBook)) {
    if (wantToSaveChanges) {
        ABAddressBookSave(addressBook, &error);
    } else {
        ABAddressBookRevert(addressBook);
    }
}

if (error != NULL) {/*... handle error ...*/}

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. 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, people and groups are stored as `ABRecordRef` objects. The function `ABRecordGetRecordType` returns `kABPersonType` if the record is a person, and `kABGroupType` if it is a group. Developers familiar with the Address Book frameworks on Mac OS should note that there are not `ABPersonRef` or `ABGroupRef` objects. Both person objects and group objects are instances of `ABRecordRef`.

Important: Record objects cannot be passed across threads safely. Instead, you should pass the corresponding record identifier. See "Using Record Identifiers" (page 25) 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 interacting directly with person records, see "Working with Person Records" (page 25).

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 `ABGroupCopyArrayOfAllMembersWithSortOrdering` or `ABGroupCopyArrayOfAllMembers`, which return a `CFArray` of `ABRecordRef` objects with and without sorting.

For more information about functions related to directly editing group records, see "Working with Group Records" (page 26).

Properties

There are two basic types of properties, single-value and multivalue. Multivalue properties can be either mutable or immutable. For a list of the properties for person records, see many of the sections within "Constants". For properties of group records, see "Group Properties" in *ABGroup Reference*.

Single-Value Properties

Here is a code sample that illustrates getting and setting a single-value property:

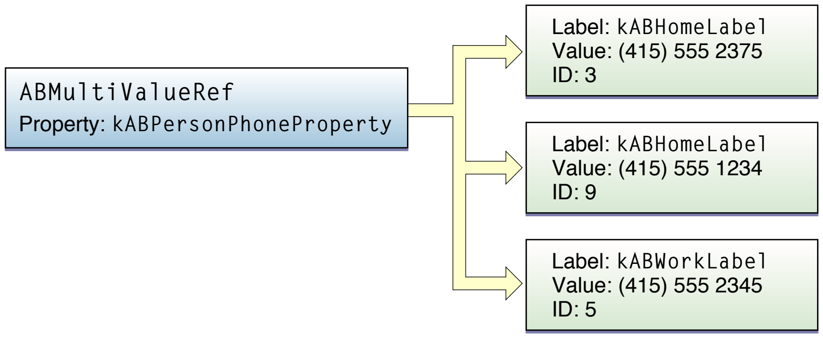
```
ABRecordRef aRecord = ABPersonCreate();
CFErrorRef  anError = NULL;

ABRecordSetValue(aRecord, kABPersonFirstNameProperty,
                 CFSTR("Katie"), &anError);
ABRecordSetValue(aRecord, kABPersonLastNameProperty,
                 CFSTR("Bell"), &anError);
if (anError != NULL) { /* ... handle error ... */}

/* ... */

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

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 the figure below 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 identifier. Note the two home phone numbers and the unique IDs

Figure 2-1 Multivalue properties



The following functions let you read the contents of multivalue properties:

- `ABMultiValueCopyLabelAtIndex` and `ABMultiValueCopyLabelAtIndex` copy individual values.
- `ABMultiValueCopyArrayOfAllValues` copies all of the values into a `CFArray`.
- `ABMultiValueGetIndexForIdentifier` and `ABMultiValueGetIdentifierAtIndex` convert between indices and multivalue identifiers.

To keep a reference to a particular value in the multivalue property, store its identifier. You should generally avoid storing the index, because it will change if values are added or removed. The identifier is guaranteed to stay the same, except across devices.

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 functions:

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

Here is a code sample that illustrates getting and setting a multivalue property:

```
ABMutableMultiValueRef multi =
        ABMultiValueCreateMutable(kABMultiStringPropertyType);
CFErrorRef anError = NULL;

// There are NULL values for the "returned identifier" because the
// multivalue identifier of the new value isn't needed in this example.
bool didAdd = ABMultiValueAddValueAndLabel(multi, @"(555) 555-1234",
                      kABPersonPhoneMobileLabel, NULL)
           && ABMultiValueAddValueAndLabel(multi, @"(555) 555-2345",
                      kABPersonPhoneMainLabel, NULL);
if (didAdd != YES) { /* ... handle error ... */}

ABRecordRef aRecord = ABPersonCreate();
ABRecordSetValue(aRecord, kABPersonPhoneProperty, multi, &anError);
if (anError != NULL) { /* ... handle error ... */}
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 stored 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*. Here is a code sample that 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(NULL,
                                           (void *)keys,
                                           (void *)values,
                                           5,

&kCFCopyStringDictionaryKeyCallBacks,
                                        &kCFTypeDictionaryValueCallBacks);

    // add the street address to the person record
    ABMultiValueIdentifier identifier;
    ABMultiValueAddValueAndLabel(address, aDict, kABHomeLabel, &identifier);
    CFRelease(aDict);

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

    CFRelease(address);
```

Interacting Using UI Controllers

The Address Book UI framework provides view and navigation controllers 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. In this chapter, you will learn about the controllers that are provided and how to use them.

The Address Book UI framework provides four controllers:

`ABPeoplePicker-NavigationController`	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.

Note: You should not need to subclass these controllers. The expected way to modify their behavior is by your implementation of their delegate.

To use these controllers, you must set a delegate for them which implements the appropriate delegate protocol. 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 2.0 Programming Language*.

Having the User Choose a Person Record

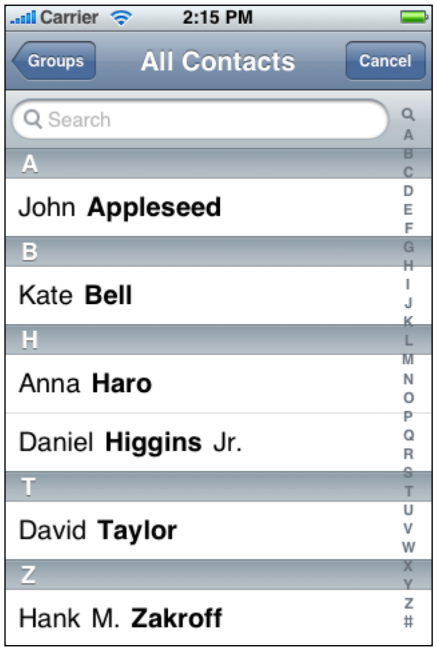
The `ABPeoplePickerNavigationController` class allows users to browse their list of contacts and select a person and, at your option, one of that person's properties. To use it, do the following:

1. Create and initialize an instance of the class.

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

3. Present the people picker as a modal view controller using the `presentModalViewController:animated:` method of the current view controller. This will make the people-picker navigation controller temporarily cover the current view controller.

Figure 3-1 People Picker Navigation Controller



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. You should tell it to continue if you want the user to choose a specific property of the selected person; otherwise you should tell it not to continue, and dismiss the picker. After the user selects a property, the people picker calls the method `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:property:identifier:` of the delegate to determine if it should continue. You should tell it to continue if you want to the people picker to perform the default action (dialing a phone number, starting a new email, etc.) for the selected property. Otherwise, you should tell it not to continue, and dismiss the picker.

The application you wrote in "Quick Start Tutorial" (page 9) contains an example of how to use this controller.

Displaying and Editing a Person Record

The `ABPersonViewController` class displays a record to the user. If you set `allowsEditing` to `YES`, the user can edit the record. You can change the properties that it displays by setting its `displayedProperties` property to the appropriate array of `ABPropertyType` values. To use this controller, create and initialize an instance of the class and set its `displayedPerson` property and delegate. The delegate must adopt the `ABPersonViewControllerDelegate` protocol. Then push the view controller onto the current navigation controller's stack.

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

Figure 3-2 Person view controller—Displaying with editing allowed

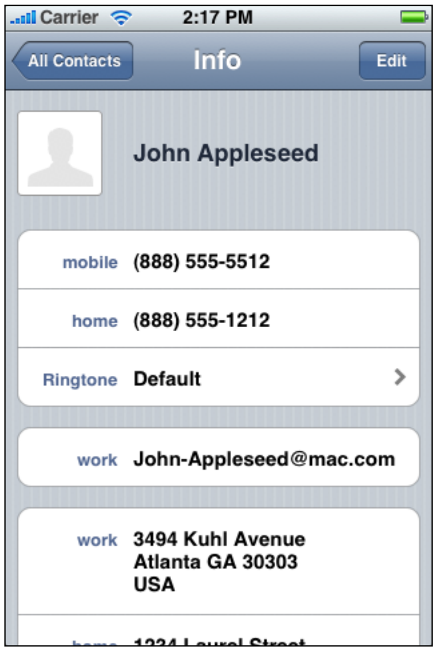
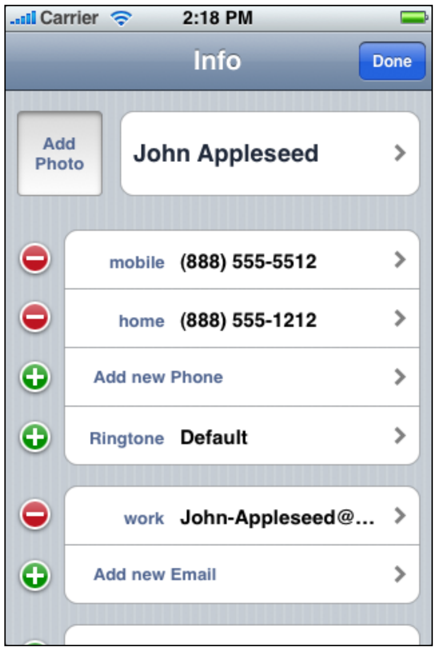


Figure 3-3 Person view controller—Editing



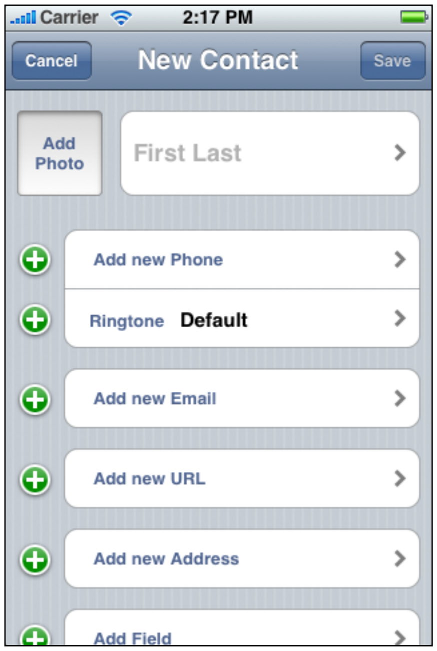
When the user taps on a property in the view, the person view controller calls the `personViewController:shouldPerformDefaultActionForPerson:property:identifier:` method of the delegate to determine if the default action for that property should be taken. For example when the user taps on a phone number, the delegate determines if the phone should dial that number.

Creating a New Person Record

The `ABNewPersonViewController` class allows users to create a new person. It is used much like a person view controller. Create and initialize a new instance of the class, and set its delegate. The delegate must adopt the `ABNewPersonViewControllerDelegate` protocol. You can (optionally) set the value of `displayedPerson` to fill in properties, or `parentGroup` to put the person in a group. Then push the view controller onto the current navigation controller's stack.

Important: New-person view controllers must be used with a navigation controller and presented modally in order to function properly.

Figure 3-4 New-person view controller



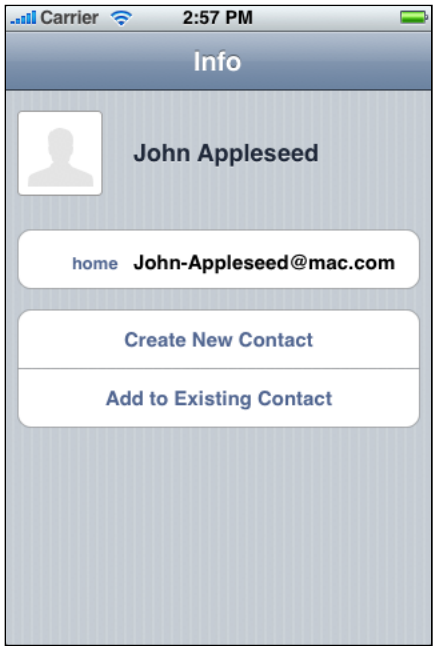
When the user taps the Save or Cancel button, the new-person view controller calls the method `newPersonViewController:didCompleteWithNewPerson:` of the delegate, with the resulting person record. If the user saved, the new record is first added to the address book. If the user cancelled, the value of `person` will be `NULL`. The delegate should also dismiss the new-person view controller.

Creating a New Person Record from Existing Data

Sometimes users have information they want to add to an existing person record or use to create a new person record. To do this, use the `ABUnknownPersonViewController` class. First, create a new person record and set the properties that are already known. Then create an unknown-person view controller, set the `displayedPerson` to the new person you just created, and set the delegate. The delegate must adopt the `ABUnknownPersonViewControllerDelegate` protocol. Then push the unknown-person view controller onto the navigation controller's stack. If you set `allowsAddingToAddressBook` to `YES`, the user can add a new contact or merge the information into an existing contact.

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

Figure 3-5 Unknown-person view pontroller



After the user has finished creating a new contact or adding the properties to an existing contact, the unknow-person view controller will call the method `unknownPersonViewController:didResolveToPerson:` of the delegate with the resulting person record. If the user cancelled, the value of `person` will be `NULL`.

Interacting Directly with the Address Book Database

Many common interactions with the Address Book database depend on user interaction. However, in some cases it's appropriate for the program to interact with the data directly. There are several functions in the Address Book framework that provide this ability.

It is important to use these functions only when they are appropriate, in order to provide a uniform user experience. Rather than using these functions to create customized view or navigation controllers, your program should call the provided view or navigation controllers whenever possible. For more information on this, see "Interacting Using UI Controllers" (page 19).

It is also worth a reminder 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 will always point to the same record unless that record is deleted. For that reason, record identifiers are the appropriate way to keep a reference to a person or group in your application's data. They can also be safely passed between threads. However, they are not guaranteed to remain the same across devices.

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`.

Working with Person Records

For basic information on person records and how they store data, see "Person Records" (page 14).

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 in the Address Book database: by name using the function `ABAddressBookCopyPeopleWithName` and by record identifier using the function `ABAddressBookGetPersonWithRecordID`.

To sort an array of people, use the function `CFArraySortValues` with the function `ABPersonComparePeopleByName` as the comparator, and a context of 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);
```

Alternatively, the array can be sorted using Objective-C as follows:

```
[allPeopleMutable sortUsingFunction:ABPersonComparePeopleByName
                            context:(void*)sortOrdering];
```

Working with Group Records

For basic information on group records and how they store data, see "Group Records" (page 15).

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`.

Document Revision History

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

Date	Notes
2008-10-15	Added example code for working with street addresses. Other minor changes throughout.
2008-09-09	Minor update for iPhone OS 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.