Address Book Programming Guide

2006-04-04

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

Simultaneously published in the United States and Canada.

Contents

Tables and Listings

Address Book Programming Guide

The Address Book is a centralized database for contact and other personal information for people. Applications that support the Address Book framework share this contact information with other applications, including Mail and iChat. Both Carbon and Cocoa applications can access users' address books. This topic covers key Address Book framework concepts and some operations you can perform with address books.

Important: If your application uses the Sync Services and Address Book frameworks together, then you should not use Sync Services to sync data shared with the Address Book Framework. The Address Book Framework already syncs its records with Sync Services, so applications sharing the Address Book data do not have to (and should not) sync those records. The results are unpredictable and may result in data loss, if you attempt to sync the same data as the Address Book Framework.

The Address Book framework consists of two APIs: one for C, the other for Objective-C. While both API are equally functional, the majority of the code samples in this document are printed in Objective-C only. Where it is appropriate, this document will address fundamental differences between the two API, but will not list sample code in both languages. Both API are similar in syntax and conventions, and mapping the Objective-C sample code to its C counterpart can be done easily using the Address Book Reference for C. Developers using the C API should also refer to "Using Address Book from C" (page 31).

Who Should Read This Document ?

This topic is designed for any Cocoa or Carbon developer who wants to leverage the abilities of the Mac OS X Address Book in their application. You will not only be able to access a user's address book data, but also design and implement your own properties and actions for the data.

It is expected that you are already familiar with Xcode and the basics of either Cocoa or Carbon development.

Organization of This Document

The topic contains the following articles:

- "About the Address Book" (page 9) describes what's in the Address Book database and what you can do with it.
- "Managing Address Book Records " (page 11) describes how to add and remove people and groups, how to arrange people into groups, and how to find the record for the logged-in user.
- "Accessing Address Book Records" (page 13) describes how to access data in a person or group record.
- "Searching an Address Book" (page 19) describes how to perform searches on a user's address book.
- "Using Address Book Groups as Distribution Lists" (page 23) describes how to set up a group so you can use it as a mailing list, or other type of distribution list.
- "Adding Properties to Address Book Records" (page 25) describes how to customize an address book for your own applications by adding properties to it.
- "Creating and Using Address Book Action Plug-ins" (page 27) describes how to create action plug-ins which allow users to perform custom actions on address book data viewed within the Address Book application.
- "Importing and Exporting Address Book People and Groups" (page 29) describes how to import and export person records by using the vCard standard.
- "Using Address Book from C" (page 31) contains special information for those using the Address Book C API.

About the Address Book

The Address Book is a centralized database for contact and other personal information for people. Users need to enter personal information about themselves and their friends only once instead of entering it repeatedly whenever the information is used. Applications that support the Address Book framework share this contact information with other applications, including Apple's Mail and iChat. Both Carbon and Cocoa applications can access it.

This database contains people's names, street addresses, email addresses, phone numbers, home pages, and more. Your applications can use this data as it is or extend it to include information specific to your applications. Every user on the computer has one and only one address book. Every application shares the address book for the currently logged-in user.

Basic Address Book Concepts

This section details the basic information that every developer using the Address Book and its associated API should know. It will help you get started with the remainder of the content in this document.

The Address Book contains two fundamental sorts of records, and they are what you would expect of an address book: ABPerson, for individuals, and ABGroup, for groups. Both are subclasses of the same root class ABRecord, and they can be used interchangeably in some places.

An ABPerson record contains such properties as the person's name, company, addresses, email addresses, phone numbers, instant messaging IDs, and a comments field.

An ABGroup object can contain any number of people and other groups. For example, let's say you are the CEO of two companies, Acme Co. and Ajax Inc. You logically would have an Acme employees group and an Ajax employees group, containing both companies' respective employees (in the form of ABPerson records). You could then set up a Professional group that includes the Acme group, the Ajax group, and some additional people who aren't in either group. A person can be in any number of groups.

Each group and person has a unique identifier that's set when the record is created. It's guaranteed never to change even if a user changes the group's or person's name or other information. Use this identifier if your application needs to store a reference to a group or person.

The groups and people are stored in an extensible form. As such, you can add custom properties to Address Book records that other applications will ignore, without worrying about data corruption or usability issues.

Some of these properties can contain multiple values. For example, a person can have any number of street addresses, phone numbers, and email addresses. A user can also specify that one of those multiple values is the primary value. For example, a user can specify that email be sent to one person's work address and another person's home address unless otherwise specified. For any such property that contains multiple values, the Address Book framework uses the ABMultiValue class. For more information, see "Using Multivalue Lists" (page 14) in "About the Address Book" (page 9).

Searching the address book of a user, of course, is vitally important. Address Book manages individual search queries using an ABSearchElement object, instances of which can be created using class methods of ABGroup and ABPerson. See "Searching an Address Book" (page 19) for a detailed look at searching Address Book records. This does have one important implication—since the ABSearchElement objects are created using ABPerson and ABGroup, a custom subclass of ABRecord will not contain the required methods to create such an object. For this reason, Apple advises against creating subclasses of ABRecord.

Advanced Address Book Concepts

This section explains some of the advanced features of Address Book, but may not be relevant to some developers.

The Address Book framework provides transparent record locking. If two applications try to change the same record simultaneously, the application that tried to change it last will succeed. The database will not be corrupted.

The Address Book does not provide any security above what's provided by Mac OS X. Anyone who has read and write access to a user's home folder can also read and write that user's address book. For that reason, the Address Book may not be an appropriate place to store confidential information, such as credit card numbers.

The Address Book API provides localized versions of the built-in property names and labels. If you add properties or labels, you must provide your own way for localizing them.

Managing Address Book Records

You can manage the people and groups within a user's Address Book. This document explains how get the user's address book, add and remove people and groups from that address book, manage groups, find the record that corresponds to the logged-in user, and save your changes.

Accessing the Address Book

To get the address book for the currently logged-in user, use the ABAddressBook method `sharedAddressBook`. If you call these procedures more than once or try to create a new address book, you get a pointer to the shared address book.

Adding and Removing People and Groups

Adding a new person or group takes two steps: creating the appropriate record and then adding it to the user's address book.

First, create the person or group. You must allocate and initialize the respective ABPerson or ABGroup object.

Second, add the person or group to the Address Book using the ABAddressBook method `addRecord:`.

To remove a person or group, use the ABAddressBook method `removeRecord:`.

Managing Groups

The Address Book lets you add people and subgroups to groups, as well as find out all groups that a person or subgroup is in.

To add and remove people from a group, use the methods `addMember:` and `removeMember:`. To get a list of all the groups a person is in, use the method `parentGroups`.

You can also add groups to a group. For example, a user could have a group called “Pet Lovers” that contains the groups “Dog Lovers” and “Cat Lovers.” To add and remove groups from another group, use the methods `addSubgroup:` and `removeSubgroup:`. The Address Book will not let you create a circular dependency. For example, if “Dog Lovers” is a subgroup of “Pet Lovers,” then “Pet Lovers” cannot be a subgroup of “Dog Lovers.” To get a list of all groups that another group is a subgroup of, use the method `parentGroups`.

To get lists of what’s in a group, use the methods `members` and `subgroups` or the functions `ABGroupCopyArrayOfAllMembers` and `ABGroupCopyArrayOfAllSubgroups`.

Accessing the User’s Record

The user can specify a record that contains information about himself or herself. That lets your application find the name, address, or phone number of the logged-in user, so you can use it when filling out forms, for example. To get the logged-in user’s record, use the ABAddressBook method `me`. To set the logged-in user’s record, use the ABAddressBook method `setMe:`.

Saving Your Changes

When you modify the Address Book database, those changes are made in memory, and not to the database itself. Unless you save those changes, they will be lost.

To save your changes to the address book, use the ABAddressBook method `save`. To test whether there are unsaved changes to the address book, use the ABAddressBook method `hasUnsavedChanges`.

An Example

This brief Objective-C example adds a person named John Doe to the current user’s address book. Take note of how the code accesses the shared address book and how it allocates a new ABPerson object. Also note the properties used (in this case, just our subject’s first name and last), and the final save, which sends the changes to the address book:

```
ABAddressBook *addressBook;
ABPerson *newPerson;

addressBook = [ABAddressBook sharedAddressBook];

newPerson = [[[ABPerson alloc] init] autorelease];

[newPerson setValue:@"John"
    forProperty:kABFirstNameProperty];

[newPerson setValue:@"Doe"
        forProperty:kABLastNameProperty];

[addressBook addRecord:newPerson];
[addressBook save];
```

Accessing Address Book Records

Once you have a record, you can retrieve the data within it. This article shows you how the data is organized and how to access it. It shows how to access properties from a records property list, how to handle properties that can have more than one value (such as addresses and phone numbers), how to get localized names for properties and labels, and how to associate a picture with a person.

Using Property Lists

Both groups and people store their data in property lists. This lets your application add properties to the Address Book records that other applications will ignore. See "Adding Properties to Address Book Records" (page 25).

To get data from them, such as a group's description or a person's first name, use the method `valueForProperty:` method or the C function `ABRecordCopyValue`. For example, to get the first name for `aPerson`, use

```
[aPerson valueForProperty:kABFirstNameProperty];
```

To set data, use the `setValue:forProperty:` method. For example, to set the name of `dataGroup`, use

```
[aGroup setValue:@"Book Club" forProperty:kABGroupNameProperty];
```

To find out the names of the properties for ABPerson and ABGroup, refer to the following documentation list:

Table 1 Documentation list for property list constants

Documentation	Class	Language
Constants	ABGroup	Objective-C
Constants	ABPerson	Objective-C
Constants	ABGroup	Procedural C
Constants	ABPerson	Procedural C

Other properties can be found in `ABGlobals.h` for Objective-C or `ABGlobalsC.h` for procedural C.

Using Multivalue Lists

Many properties can have multiple values. For example, a person can have several addresses, including work, home, summer home, and mailing addresses. These properties are stored as multivalue lists, of type ABMultiValue or ABMultiValueRef. Each item in a multivalue list has a numeric index, a unique identifier, a string label (such as "Home" or "Work"), and a value. Every value in the multivalue list must be of the same type. The label does not need to be unique; after all, someone could have more than one home or work address. You access the items with the numeric index. To add an item to a multivalue list, use the method `addValue:withLabel:`. To retrieve an item, use the methods `valueAtIndex:` and `labelAtIndex:`.

If you want to save a reference to a specific value, use the unique identifier. The numeric index may change as the user adds and removes values, but the identifier is guaranteed never to change. To get the unique identifier for a value at a particular index, use the method `identifierAtIndex:`. To get the index for a identifier, use the method `indexForIdentifier:`.

Each multivalue list also has a primary value, which is the item the user most strongly associates with that person. For example, friends may have both home and work addresses, but the home address is their primary address. And coworkers may have both home and work phone numbers, but the work number is their primary number. To get the identifier for a multivalue list's primary value, use the method `primaryIdentifier`. To set the multivalue list's primate value, use the method `setPrimaryIdentifier:`.

Associating a Picture With a Person

You can associate a picture that identifies a person in your Address Book database. Because these pictures are not stored in the same way as the other Address Book, you need to use different methods to access them.

To associate a TIFF to a person, use the method setImageData: or the function `ABPersonSetImageData`. To get the TIFF data for person's image, use the method `imageData`. Note that both NSImage and QuickTime have functions for converting the data into a TIFF file.

Images are located by Address Book through a specific search heirarchy, in this order:

1. Check for an image set specifically by the user.
2. Check Directory Services for the local user's login picture.
3. Check for an image in /Network/Library/Images/People/*email*, where *email* is the user's primary e-mail address.
4. Check for an image from the user's .Mac account, first against the cache at ~/Library/Caches/com.apple.AddressBook/*email* and then against the .Mac servers.

Getting Localized Names for Properties and Labels

You can find the localized name for any of the property names and labels that are in the header files `ABGlobals.h` and `ABGlobalsC.h`. In C, the function `ABCopyLocalizedPropertyOrLabel` returned a name that's localized for the user's selected language.

If you want to localize names for the properties and labels that you create, you must handle it yourself. The Address Book framework does not have support for this.

An Example

Listing 1 (page 15) is an Objective-C code sample that retrieves the country for the primary address of the logged-in user. If the country is a null string, it sets the country to USA.

Listing 1 Changing a Person's Address, in Objective-C

```
ABPerson *aPerson = [[ABAddressBook sharedAddressBook] me];
ABMutableMultiValue *anAddressList =
    [[aPerson valueForProperty:kABAddressProperty] mutableCopy];
int primaryIndex =
    [anAddressList indexForIdentifier:[anAddressList primaryIdentifier]];
NSMutableDictionary *anAddress =
    [[anAddressList valueAtIndex:primaryIndex] mutableCopy];
NSString *country =
    (NSString*) [anAddress objectForKey:kABAddressCountryKey];
if ([country isEqualToString:@""]) {
    [anAddress setObject:@"USA" forKey:kABAddressCountryKey];
    [anAddressList replaceValueAtIndex:primaryIndex withValue:anAddress];
    [aPerson setValue:anAddressList forProperty:kABAddressProperty];
    [[ABAddressBook sharedAddressBook] save];
}
```

Using the People Picker

The People Picker is a portion of the Address Book API that provides easy and quick access to the contents of a user's address book from any application. It offers a searchable, selectable list of people and groups that can be customized for your application.

Important: The People Picker was introduced in Mac OS X version 10.3 and is only available to applications on systems running that version or later.

There are two ways to integrate the People Picker into a Cocoa application. The first is to use Interface Builder and the Address Book palette. The palette is not included in the standard list of Interface Builder palettes. To add it, click the Palettes tab in Xcode's Preferences, click the Add button, and select `ABPalette.palette`, located at `/Developer/Extras/Palettes/ABPalette.palette`.

You can drag the ABPeoplePickerView from the AB palette to your window, and use the Interface Builder Info window to set many of the attributes of the People Picker view. These include the columns the view should display, whether or not multiple selections are allowed, whether or not group selections are allowed, and the autosave name for the view. These changes are reflected immediately, even within Interface Builder. Using the Test Interface feature of Interface Builder, a People Picker view will display the address book of the logged-in user.

The other method is to create and modify the People Picker view programmatically. To do so, you must design the enclosing window yourself with the People Picker view and make the appropriate connections to your controller class in the nib file. The most basic People Picker window needs a custom view object. This custom view should be set to a custom class of `ABPeoplePickerView`. This will require you to include `AddressBook.h` and `AddressBookUI.h` in your controller, make a subclass of NSView called `ABPeoplePickerView`, and create an outlet of that type from your controller to the custom view.

With just this People Picker view connected and no other code inserted, you get a working view of the Address Book, displaying the current user's groups and people, as well as a search bar to narrow the results.

The People Picker API provides a host of methods and functions to access the address book data, to change the layout and construction of the People Picker view, and even to activate the Address Book application to perform tasks such as editing records.

It should be noted that the Carbon People Picker is only available in a window form and cannot be used as a custom HIView. It must be created as an `ABPickerRef` with `ABPickerCreate()` and made visible with `ABPickerSetVisibility()`. The code would look like this:

```
ABPickerRef peoplePicker = ABPickerCreate();
ABPickerSetVisibility(peoplePicker, TRUE);
```

For Cocoa applications, the People Picker also provides methods for using autosave data, so that it can retain the filter selections and column positions. See the documentation on `autosaveName:` and `setAutosaveName:`.

People Picker Example

This example would be placed in the window controller for the People Picker. Carbon developers should refer to theAddress Book Reference for C to construct the analogue for their applications. Most of the functions are named similarly. In lieu of notifications, you will need to register event handlers to handle changes to the window.

This sample incorporates the fundamentals of using the People Picker. It assumes you have created a window in your nib file with an `ABPeoplePickerView` object, a NSTextField outlet called `nameFieldLabel`, and a NSImageView outlet called `personImage`.

```
#import "PickerController.h"
@implementation PickerController
- (void)awakeFromNib
{
    NSNotificationCenter* center;
    center = [NSNotificationCenter defaultCenter];

    //Here we set up a responder for one of the four notifications,
```

```
    //in this case to tell us when the selection in the name list
    //has changed.
    [center addObserver:self
            selector:@selector(recordChange:)
            name:ABPeoplePickerNameSelectionDidChangeNotification
            object:peoplePicker];

    //Here we disallow multiple selections in the name list
    [peoplePicker setAllowsMultipleSelection:NO];

    //Here we add the e-mail and telephone properties to the
    //view. By default, the People Picker displays only the
    //Name column.
    [peoplePicker addProperty:kABEmailProperty];
    [peoplePicker addProperty:kABPhoneProperty];
}

//This is the responder for the notification we registered for
- (void)recordChange:(NSNotification*)notif {
    NSImage *personImage;
    NSString *personName;
    NSArray *array;

    array = [peoplePicker selectedRecords];
    ABPerson *person = [array objectAtIndex:0];
    personImage = [[NSImage alloc] initWithData:[person imageData]];
    personName  = [NSString stringWithFormat:@"%@ %@",
                    [person valueForProperty:kABFirstNameProperty],
                    [person valueForProperty:kABLastNameProperty]];

    [imageView setImage:personImage];
    [nameField setStringValue:personName];

    [personImage release];
}

@end
```

Searching an Address Book

You can quickly search a user's address book, using arbitrarily complex criteria. The address book is fully indexed, automatically. And the Address Book framework's searching features let you easily create complex searches. For example, you can search for all people named "Smith," or for all people who work at Acme and live in San Francisco, or who work at Ajax and live in Seattle.

Here's how you create a search query. For each property you want to search on, create a search element. If you want to search on multiple properties, combine the search elements together with Boolean operators to create a complex search element. Then search the Address Book with that simple or complex search element.

Creating a Search Element for a Single Property

To create a search element for a person, use the ABPerson class method `searchElementForProperty:label:key:value:comparison:`. To create a search element for a group, use the ABGroup class method `searchElementForProperty:label:key:value:comparison:`. These procedures take the following arguments:

- *`property`* is the name of the property to search on, such as `kABAddressProperty` or `kABLastNameProperty`. It cannot be `nil`. For a full list of the properties, see `ABGlobals.h`.
- *`label`* is the label name for a multivalue list, such as `kABAdressHomeLabel`, `kABPhoneWorkLabel`, or a user-specified label, such as `"Summer Home"`. If the specified property does not have multiple values, pass `nil`. If the specified property does have multiple values, pass `nil` to search all the values.
- *`key`* is the key name for a dictionary, such as `kABAddressCityKey` or `kABAddressStreetKey`. If the specified property is not a dictionary, pass `nil`. If the specified property is a dictionary, pass `nil` to search all keys.
- *`value`* is what you're searching for. It cannot be `nil`.
- *`comparison`* specifies the type of comparison to perform. You can choose
  - To search for elements that are equal or not equal to the value, use `kABEqual`, `kABNotEqual`, or `kABEqualCaseInsensitive`
  - To search for elements that are less than or greater than the value, use `kABLessThan`, `kABLessThanOrEqual`, `kABGreaterThan`, or `kABGreaterThanOrEqual`.

- To search for elements `kABContainsSubString`, `kABContainsSubStringCaseInsensitive`, `kABPrefixMatch`, or `kABPrefixMatchCaseInsensitive`.

Creating a Search Element for Multiple Properties

To combine search elements, use the ABSearchElement class method `searchElementForConjunction:children:`. These procedures take two arguments:

- *`conjunctionOperator`* describes how to combine the search elements. It can be `kABSearchAnd` or `kABSearchOr`.
- *`children`* is an NSArray of search elements. The search elements can be a simple elements that specifies only one property, or complex elements that specifies several. This lets you create arbitrarily complex search elements. You cannot combine search elements for groups with search elements for people.

Finding Records that Match a Search Element

To search the Address Book for records that match a search element, use the ABAddressBook method `recordsMatchingSearchElement:`. These procedures return an NSArray of records.

Search Examples

Listing 1 shows a simple search. This code finds all the people whose last name is "Smith."

Listing 1 Simple Search

```
ABAddressBook *AB = [ABAddressBook sharedAddressBook];
ABSearchElement *nameIsSmith =
    [ABPerson searchElementForProperty:kABLastNameProperty
                                 label:nil
                                   key:nil
                                 value:@"Smith"
                            comparison:kABEqualCaseInsensitive];
NSArray *peopleFound =
    [AB recordsMatchingSearchElement:nameIsSmith];
```

Listing 2 shows a complex search. It searches for anyone who lives in San Francisco and works for Acme, or for anyone who lives in Seattle and works for Ajax. Note that the addresses are searched using the `kABHomeLabel` label—we only want to know if they live in the city we are searching, not if they work in the same city.

Listing 2 Complex Search

```
ABAddressBook *AB = [ABAddressBook sharedAddressBook];
```

```
ABSearchElement *inSF =
    [ABPerson searchElementForProperty:kABAddressProperty
                                 label:kABHomeLabel
                                   key:kABAddressCityKey
                                 value:@"San Francisco"
                            comparison:kABEqualCaseInsensitive];
ABSearchElement *atAcme =
    [ABPerson searchElementForProperty:kABOrganizationProperty
                                 label:nil
                                   key:nil
                                 value:@"Acme"
                            comparison:kABContainsSubStringCaseInsensitive];
ABSearchElement *inSeattle =
    [ABPerson searchElementForProperty:kABAddressProperty
                                 label:kABHomeLabel
                                   key:kABAddressCityKey
                                 value:@"Seattle"
                            comparison:kABEqualCaseInsensitive];
ABSearchElement *atAjax =
    [ABPerson searchElementForProperty:kABOrganizationProperty
                                 label:nil
                                   key:nil
                                 value:@"Ajax"
                            comparison:kABContainsSubStringCaseInsensitive];
ABSearchElement *inSFAndAtAcme =
    [ABSearchElement searchElementForConjunction:kABSearchAnd
                                        children:[NSArray arrayWithObjects:
                                            inSF, atAcme, nil]];
ABSearchElement *inSeattleAndAtAjax =
    [ABSearchElement searchElementForConjunction:kABSearchAnd
                                        children:[NSArray arrayWithObjects:
                                            inSeattle, atAjax, nil]];
ABSearchElement *inSFAndAtAcmeOrInSeattleAndAtAjax =
    [ABSearchElement searchElementForConjunction:kABSearchOr
                                        children:[NSArray arrayWithObjects:
                                            inSFAndAtAcme, inSeattleAndAtAjax,
 nil]];
NSArray *peopleFound =
    [AB recordsMatchingSearchElement:inSFAndAtAcmeOrInSeattleAndAtAjax];
```

Using Address Book Groups as Distribution Lists

An Address Book group can be used as a distribution list. For example, a user can have a Christmas Card group of all the people he or she mails Christmas cards to. Or a user can have a book club group of all the people he or she emails book club announcements to. The Address Book framework provides you with a special feature that helps you maintain distribution lists.

If a property has multiple values, like a street address or email address, it lets the user choose one as the value that this group uses. Generally, the user will want to use the value he or she has already marked as primary. But in some cases the user might want to make an exception. For example, for coworkers, the user would want their work email addresses to be their primary email addresses. But when the user notifies some of them about a book club that happens on the weekends, he or she would want to send email to their home addresses. Each group can use a different value for each person.

To choose the value in a multivalue list that a group uses, use the ABGroup method `setDistributionIdentifier:forProperty:person:`. To get a group's chosen value for a multivalue list, use the ABGroup method `distributionIdentifierForProperty:person:`. These procedures return the identifier for the value chosen for this group, or the identifier for the primary value if no value is chosen for the group.

Adding Properties to Address Book Records

You can add your own properties to the people and groups in the Address Book. For example, if you're creating a small application to manage a dog club, you could add properties to each person that specify the name and breed of that person's dog. Or if you're creating an application to manage business contacts, you could add a property that lists all the meetings and phone calls a user has had with that person. These properties are stored in the Address Book database. Applications that don't know about the new properties aren't affected by them and don't modify them.

When deciding whether to add a property to the Address Book, keep these issues in mind:

- Avoid properties for confidential information, such as credit card numbers. The Address Book does not provide any security above what's provided by Mac OS X. Anyone who has read and write access to a user's home folder can also read and write that user's address book.
- Avoid properties that are not useful for everyone in the address book. If you want to store information for just the logged-in user, consider using the NSUserDefaults or CFPreference APIs.
- Use a multivalue list if you think a person may have more than one of that property. Your new multivalue list has the same capabilities as the other multivalue lists in the address book. The user can choose a primary value in the list and can create distribution lists for it.

To add properties to every person or group, use the ABPerson or ABGroup class method `addPropertiesAndTypes:`. These procedures take a NSDictionary or CFDictionary, in which the keys are the names of the new properties and the values are their types. Note that the property names must be unique. You may want to use Java-style package names for your properties, to make sure no one else uses the same name; for example, `"org.dogclub.dogname"` or `"com.mycompany.meetinglist"`. The type can be one of five types or a multivalue list of one of those types. Here are the types:

- `kABStringProperty` or `kABMultiStringProperty`
- `kABIntegerProperty` or `kABMultiIntegerProperty`
- `kABRealProperty` or `kABMultiRealProperty` (a floating-point number)
- `kABDateProperty` or `kABMultiDateProperty` (an NSDate)
- `kABArrayProperty` or `kABMultiArrayProperty` (an NSArray)
- `kABDictionaryProperty` or `kABMultiDictionaryProperty` (an NSDictionary)
- `kABDataProperty` or `kABMultiDataProperty` (an NSData)

Creating and Using Address Book Action Plug-ins

A unique aspect of Address Book is its ability to act on data contained within a person's card. You can install your own custom plug-ins to add additional actions to a given record. An example of an existing action is the “Large Type” action, which works on any phone number entry. When selected from its rollover menu, it displays the number in large type across the screen.

The action plug-in protocol, which must be followed for Address Book to recognize the plug-in, is specified in `ABActionDelegate`. The plug-in must respond to three methods (`actionProperty`, `titleForPerson:identifier:`, `performActionForPerson:identifier:`), and may optionally respond to `shouldEnableActionForPerson:identifier:`. Note that Carbon applications must also implement the `ABActionRegisterCallbacks` function. The following table describes the action methods and functions, and what their purpose is in an Address Book action plug-in:

Table 1 Action Methods for an Address Book action plug-in

Method	Purpose
`(void)performActionForPerson:(ABPerson *)`*`person`*` identifier:(NSString *)`*`identifier`*	Performs the appropriate action for the plug-in. Each plug-in supports only one action.
`(NSString *)titleForPerson:(ABPerson *)`*`person`*` identifier:(NSString *)`*`identifier`*	Returns the title of the menu item for the action.
`(NSString *)actionProperty`	Returns the ABProperty constant that the action applies to.
`(BOOL)shouldEnableActionForPerson:(ABPerson *)`*`person`*` identifier:(NSString *)`*`identifier`*	Returns YES if the action is applicable and NO otherwise.

For a description of all the appropriate constants, refer to Address Book Reference for Objective-C. In addition, see ABActionDelegate reference for a complete description of the methods above, including how to leverage the method parameters to customize the plug-in’s action.

An example plug-in project is included with Xcode. Out of the box, it will compile into an action plug-in designed to create a rollover menu item on any phone number. When the menu item is selected, Address Book will speak the number using Mac OS X’s speech synthesis framework. To view this project and review its code, create a new Address Book Action Plug-in from Xcode’s New Project window. You will edit this pre-created template whenever you want to make a new plug-in.

Once your project is complete, you may want to change the wrapper extension from `.bundle` to something more appropriate, such as `.plugin`. This can be changed in the Styles pane of the project Inspector, and is completely optional. Do that step or not, you can build your project. The completed bundle should be placed in `~/Library/Address Book Plug-Ins` (to only use it on your user account) or in `/Library/Address Book Plug-ins` (to offer it to all users on the machine).

Importing and Exporting Address Book People and Groups

You can import and export people in the Address Book using the vCard format. To create a vCard representation of a person, use the method `vCardRepresentation`. This method creates an NSData structure that you can use in your program or save to a file.

To create a person from a vCard representation, use the method `initWithVCardRepresentation:`.

Using Address Book from C

This article contains important information for developers using Address Book's C API. For the most part, the Objective-C API has close method and syntax parity with the C API. This makes it easy to determine, for example, which function corresponds to a given Objective-C method.

There are a couple of primary differences that Carbon developers need to be aware of when using the Address Book C API:

- The Carbon People Picker comes only in window form and does not have an API for setting an accessory view. In addition, changes in selection and displayed properties are sent via Carbon Events.
- When creating an action plug-in using Carbon, your CFBundle must implement a function called `ABActionRegisterCallbacks`, which will return an `ABActionCallbacks` structure. The structure needs to be formed according to this type definition:

```
typedef struct {
    //the version of this struct is 0
    CFIndex                     version;

    //A pointer to a function that returns the AddressBook
    //property this action applies to.
    ABActionPropertyCallback    property;

    //A pointer to a function that returns the AddressBook
    //property this action applies to. Only items with labels
    //may have actions at this time.
    ABActionTitleCallback       title;

    // A pointer to a function which returns YES if the action
    //should be enabled for the passed ABPersonRef and item
    //identifier. The item identifier will be NULL for single value
    //properties. This field may be NULL. Actions with NULL enabled
    //callbacks will always be enabled.
    ABActionEnabledCallback     enabled;

    //A pointer to a function which will be called when the user
    //selects this action. It's passed an ABPersonRef and item
    //identifier. The item identifier will be NULL for single
    // value properties.
    ABActionSelectedCallback    selected;

} ABActionCallbacks
```

Otherwise, the Objective-C and C APIs act quite similar.

To access the user's shared address book from Carbon, you need to set an `ABAddressBookRef` to the return value of `ABGetSharedAddressBook`:

```
ABAddressBookRef addressBook = ABGetSharedAddressBook();
```

Compare this with the same line, but from a Cocoa Objective-C application:

```
ABAddressBook *addressBook = [ABAddressBook sharedAddressBook];
```

Notice the similarity in the method and function names. Most of the sample code in this document is written in Objective-C, except where required to explain differences in the API. However, you can see that the mapping between the two APIs is easy to follow.

Let's take an example from "Searching an Address Book" (page 19). Listing 1 (page 32) searches for anyone named Smith in the current user's address book and returns an array of results:

Listing 1 Simple Search, in Objective-C

```
ABAddressBook *AB = [ABAddressBook sharedAddressBook];

ABSearchElement *nameIsSmith =
    [ABPerson searchElementForProperty:kABLastNameProperty
                                 label:nil
                                   key:nil
                                 value:@"Smith"
                            comparison:kABEqualCaseInsensitive];

NSArray *peopleFound =
    [AB recordsMatchingSearchElement:nameIsSmith];
```

In Listing 2 (page 32) you see that very same code segment, but written using the C API:

Listing 2 Simple Search, in C

```
ABAddressBookRef AB = ABGetSharedAddressBook();

ABSearchElementRef nameIsSmith =
    ABPersonCreateSearchElement(kABLastNameProperty,
                    NULL,
                    NULL,
                    CFSTR("Smith"),
                    kABEqualCaseInsensitive);

CFArrayRef peopleFound =
    ABCreateArrayOfMatchingRecords(AB, nameIsSmith);
```

Look familiar? For more details about integrating the Address Book into your Carbon applications, refer to Address Book Reference for C.

Document Revision History

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

Date	Notes
2006-04-04	Made minor editorial corrections throughout.
2005-04-29	Made minor sample code changes. Added important note about syncing with Sync Services to the introduction.
2004-04-21	Added major updates. New sections include "Creating and Using Address Book Action Plug-ins" (page 27), "Using the People Picker" (page 15), and "Using Address Book from C" (page 31) for Carbon developers. Other sections have new sample code and more detailed content.
2003-10-30	Made minor correction in "Search Examples" (page 20).
2003-08-21	Added revision history, which records changes to the content of *Address Book*.