AboutBox: Creating a Framework With Project Builder

This tutorial shows how to create a project that builds both a framework and an application that uses that framework. The framework contains a function that displays a dialog box, a resource file with that dialog box, and a header file that declares the function. To do this, you'll create a project that builds an application, then create a framework it will use. Along the way, you'll learn a little on how Mac OS X stores software configuration information.

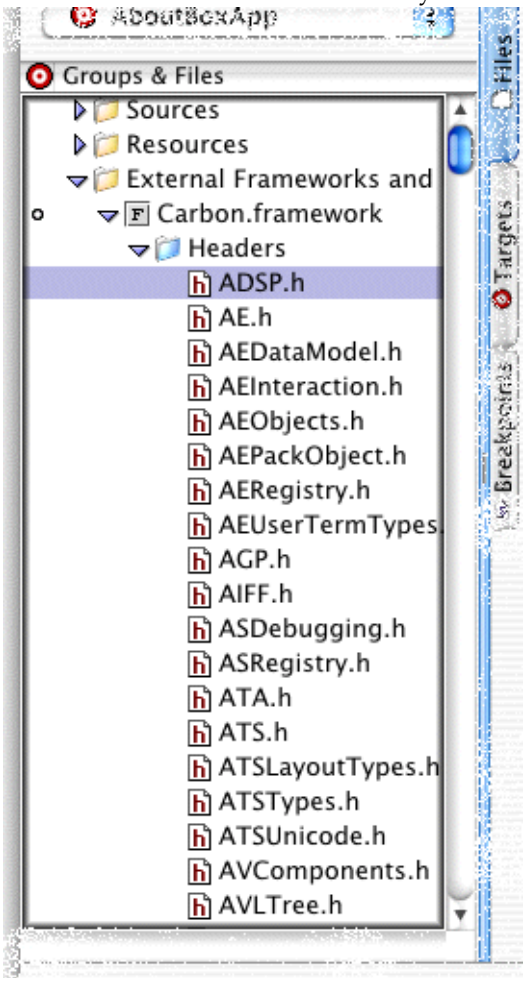
This tutorial assumes that you're familiar with Mac OS programming.

Create the Project

Choose File > New Project. Select Carbon Application, and click Next. Name the project `AboutBoxApp`, choose a location, and click Finish. Project Builder creates a new project and opens its project window. The project contains sample files you can compile and run without change. Later, you'll excerpt a function from these sample files that displays an About box, and build a framework around it.

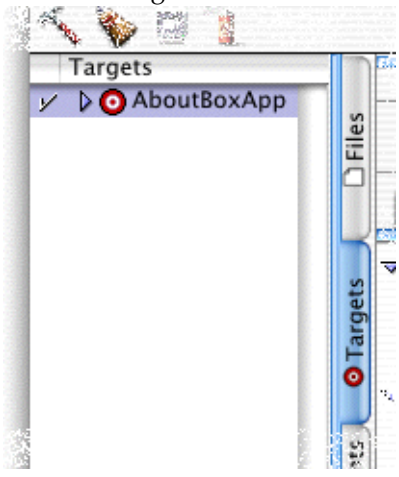
Take a moment to look at the framework and the target already in the project. Later, you'll create a new target and framework.

The Carbon framework contains all the Mac OS functions that are Carbon-compliant. To open the Carbon framework, click the disclosure triangle next to it. Project Builder displays a folder of headers. When you open that folder, you can see all of Carbon's header files. Your source files can include any of these files, and Project Builder will know to search for them within this framework. A framework contains a shared library and all the resources and headers files it uses.

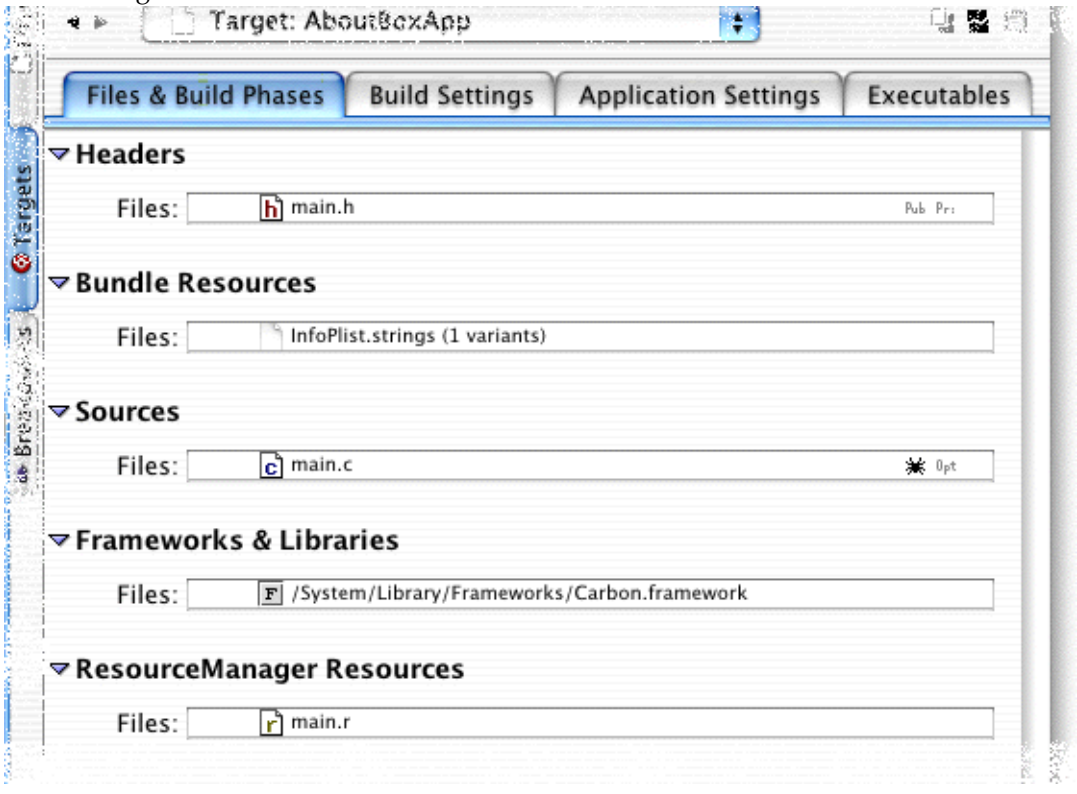


The framework you'll be creating in this tutorial contains not only its header file but also its own resource file. To use the framework in a new application, you need to add only one file to your project, instead of adding separate library, resource, and header files.

The AboutBoxApp target builds a simple Carbon application. To look at this target, click the Targets tab and click AboutBoxApp.



Click the Files & Build Phases tab. This displays the files list for the AboutBoxApp target.



This list contains a subset of the files in the project: only the files this target needs. Notice that the files fall into four categories according to how the build system handles them:

- Headers: Files that aren't compiled, but that the target needs to manipulate somehow, such as copying them into a framework.
- Bundle Resources: Files to copy into the product's resource folder. These are usually files of localized strings, Nib files, sounds, and pictures. Note that if a file needs to be compiled by Rez, it belongs in the ResourceManager Resources category.
- Sources: Files that need to be compiled, such as C++, Objective-C, or Java source files.

- Frameworks & Libraries: Files of already compiled code the product needs to link against.
- ResourceManager Resources: Files to merge into the product's resources. These are usually Rez (.r) files and resource (.rsrc) files.

The other tabs contain options that control how Project Builder builds the target. You won't need to change their settings in this tutorial.

Create and Build the New Framework

In this section, you'll create and build a new framework that displays an About box.

1. "Create the Framework Target" (page 19)
2. "Add Any Necessary Frameworks" (page 19)
3. "Add the Source, Header, and Resource Files" (page 20)
4. "Mark the Public Header Files" (page 26)
5. "Assign a Bundle Identifier and Executable Name to the Framework" (page 27)
6. "Build the Framework" (page 28)
7. "Regroup the Files" (page 28)

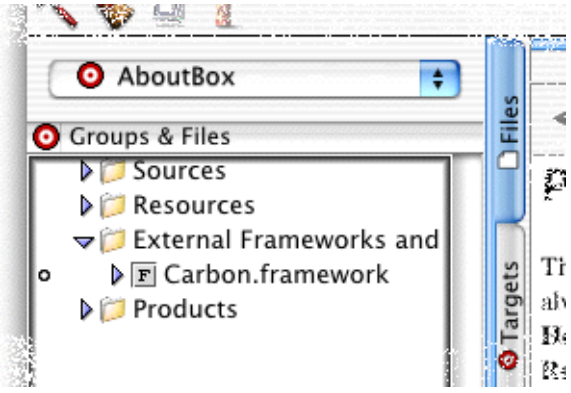
Create the Framework Target

Choose Project > New Target. Select Framework as the project type, and name it AboutBox. This creates a new target that builds a framework named AboutBox.

Add Any Necessary Frameworks

Because the AboutBox target will use functions from Carbon, you must add the Carbon framework to it. Just click the Files tab, select AboutBox in the pop-up menu above the Files list, and click beside `Carbon.framework`. A circle appears beside it to

show that it's now part of the target that's displayed in the pop-up menu above the files list.



Add the Source, Header, and Resource Files

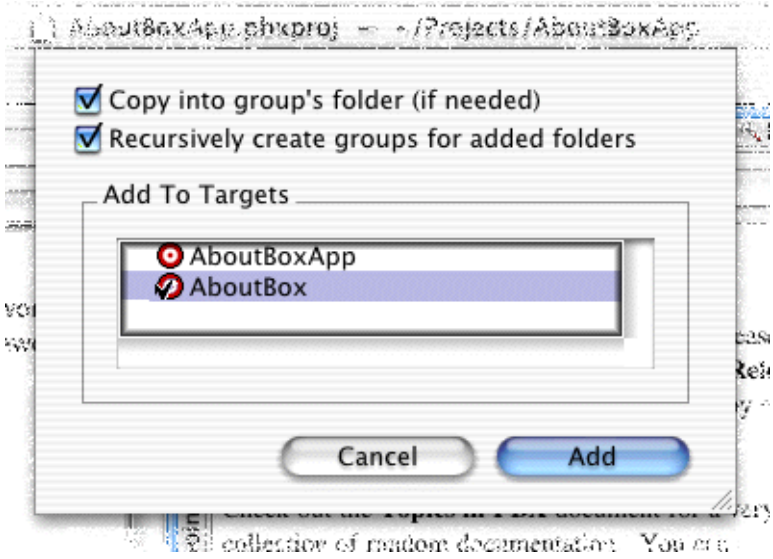
- Choose Project > Add Files, and select `AboutBox.c`, `AboutBox.h`, and `AboutBox.r`.

  These files should be in the same folder as this tutorial (`/System/Documentation/Developer/DeveloperTools/ProjectBuilder/AboutBox/`).

- Copy the files into the folder, and add the files to the AboutBox target.

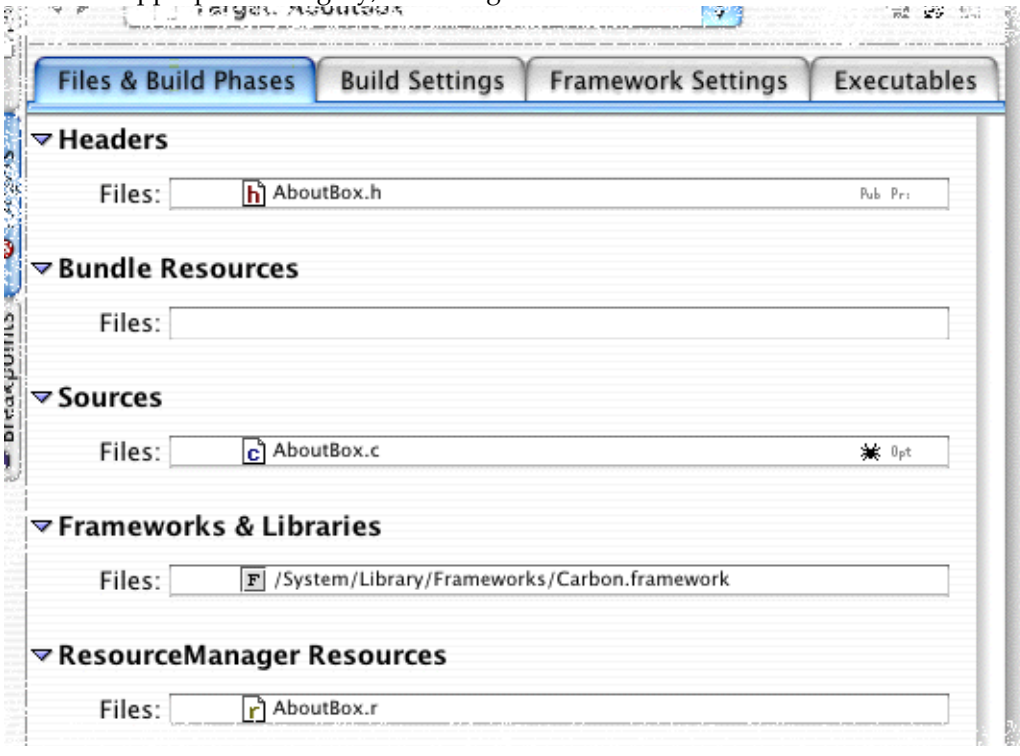
  Select "Copy into group's folder." In the Add to Targets box, make sure that AboutBox is checked and AboutBoxApp is not checked. The setting of

"Recursively create groups for added folders" doesn't matter since you are not adding folders.



Project Builder adds these files to the project, copies them to the project's folder, and adds them to the AboutBox target.

To see the new contents of the AboutBox target, click the AboutBox target, click the Files & Build Phases tab, and open all the categories. Project Builder added each file to the appropriate category, according to the file's extension.



`AboutBox.c` defines the function `DoAboutBox`, which displays a simple dialog box with the application's name. `AboutBox.h` declares that function. And `AboutBox.r` contains the resources for the dialog box it displays. Here are the contents of `AboutBox.c`:

Listing 2-1 AboutBox.c

```
#include <MacWindows.h>
#include <Dialogs.h>
#include <CoreFoundation/CFBundle.h>
#include "AboutBox.h"

#define kAboutBox200/* Dialog resource for About box */
```

```
void DoAboutBox(void)
{
    CFBundleRef appBundle, fwBundle;
    CFStringRef cfVersionString;
    Str255 pascalVersionString;
    short ierr, globalRefNum, localRefNum;

    /* Get the application's short version string. */
    appBundle = CFBundleGetMainBundle();
    cfVersionString = (CFStringRef) CFBundleGetValueForInfoDictionaryKey(
       appBundle, CFSTR("CFBundleShortVersionString"));
    if ((cfVersionString == CFSTR("")) || (cfVersionString == NULL))
        cfVersionString = CFSTR("Nameless Application");
    CFStringGetPascalString(cfVersionString, pascalVersionString, 256,
        CFStringGetSystemEncoding());

    /* Open the framework's resource fork. */
    fwBundle = CFBundleGetBundleWithIdentifier(
        CFSTR("com.apple.tutorial.aboutbox") );
    ierr = CFBundleOpenBundleResourceFiles( fwBundle, &globalRefNum,
        &localRefNum );

    /* Display the about box (from the framework)
       with the version string (from the application). */
    ParamText(pascalVersionString,"\p","\p","\p");
    (void) Alert(kAboutBox, nil);

    /* Close the framework's resource fork. */
    CFBundleCloseBundleResourceMap(fwBundle, globalRefNum);
    CFBundleCloseBundleResourceMap(fwBundle, localRefNum);
}
```

This code makes heavy use of Mac OS X's new features for configuring software. An application or framework is a bundle, a folder of files that the Finder treats as a single unit. In this tutorial, both the application and its framework are bundles.

The first block of code retrieves the application's short version string, which is the application's name and version number. A bundle's information dictionary stores that string, as well as the location of the bundle's icon, the document types it can open, and other configuration information. A Classic Mac OS application stores this

sort of data in a variety of resources, such as the `BNDL`, `SIZE`, and `vers` resources. A Mac OS X application stores it in two places inside the bundle: an XML file called `Info.plist` and in localized string files called `InfoPlist.strings`. `Info.plist` contains information that doesn't need to be translated into different languages, such as the executable's name on the disk and the bundle's unique identifier that are used only in code. `InfoPlist.strings` contains information that does need to be translated, such as the Get Info string and short version string, both of which are seen by users. A bundle can contain several `InfoPlist.strings` files, each stored in a different localization directory, such as `English.lproj` and `Japanese.lpoj`, along with other localized resources. You'll enter the short version string for the AboutBoxApp target in "Assign a Short Version String to the Application" (page 32).

The second block of code opens the framework's resources, which contain a simple dialog box. To open the resources, which are in the framework's bundle, the framework finds the bundle with its unique identifier `"com.apple.tutorial.aboutbox"`. You'll assign that identifier to the framework in "Assign a Bundle Identifier and Executable Name to the Framework" (page 27).

The last two blocks of code display the dialog box and close the resources.

The rest of this section describes how this function works, line-by-line. If you want, you can skip to "Mark the Public Header Files" (page 26).

Retrieving the Application's Short Version String

This code is useful in any code that needs to access individual keys in a bundle's information dictionary. To see what keys are available, click a target and click its Application Settings or Framework Settings tab. To see what keys are localizable, look at the `InfoPlist.strings` file. For more information on what the keys mean, see *Software Configuration* (at `/System/Documentation/Developer/SystemOverview/SoftwareConfig.pdf`)

```
/* Get the application's short version string. */
appBundle = CFBundleGetMainBundle();                                    // 1
cfVersionString = (CFStringRef) CFBundleGetValueForInfoDictionaryKey( // 2
   appBundle, CFSTR("CFBundleShortVersionString"));
if ((cfVersionString == CFSTR("")) || (cfVersionString == NULL))      // 3
    cfVersionString = CFSTR("Nameless Application");
CFStringGetPascalString(cfVersionString, pascalVersionString, 256,    // 4
    CFStringGetSystemEncoding());
```

1. `CFBundleGetMainBundle` retrieves the main bundle, which, in this case, is the bundle for the application that's using this framework. To find another bundle, use `CFBundleGetBundleWithIdentifier`.
2. `CFBundleGetValueForInfoDictionaryKey` retrieves the value that's stored for the specified key, which is `"CFBundleShortVersionString"`. First, it looks in the `InfoPlist.strings` file for the user's region. If it can't find the value there, it looks in the `Info.plist` file
3. If the application doesn't specify a short version string, the `if` statement uses `"Nameless Application"` instead.
4. `CFStringGetPascalString` converts a Core Foundation string to a Pascal string. It's needed because `CFBundleGetValueForInfoDictionaryKey` returned a Core Foundation string, but `ParamText`, below, needs a Pascal string

Opening the Framework's Resources

This code is useful in any framework that has its own resources.

```
/* Open the framework's resource fork. */
fwBundle = CFBundleGetBundleWithIdentifier(                          // 1
    CFSTR("com.apple.tutorial.aboutbox") );
ierr = CFBundleOpenBundleResourceFiles( fwBundle, &globalRefNum,     // 2
    &localRefNum );
```

1. `CFBundleGetBundleWithIdentifier` returns a reference to the AboutBox framework, by searching for its unique identifier `"com.apple.tutorial.aboutbox"`. Later in this tutorial, you'll assign that identifier to the framework.
2. `CFBundleOpenBundleResourceFiles` opens the bundle's resources, both the global and the localized versions. Note that a bundle's resources are usually stored as a separate file inside the bundle.

Displaying the About Box

This code displays the About box and closes the resources.

```
/* Display the about box (from the framework)
   with the version string (from the application). */
ParamText(pascalVersionString,"\p","\p","\p");                       // 1
(void) Alert(kAboutBox, nil);                                        // 2
```

```
/* Close the framework's resource fork. */
CFBundleCloseBundleResourceMap(fwBundle, globalRefNum);          // 3
CFBundleCloseBundleResourceMap(fwBundle, localRefNum);
```

1. The dialog box contains a text field with the string `"^0"`. `ParamText` substitutes its argument (the application's short version string) for that string.
2. `Alert` displays the dialog box.
3. `CFBundleCloseBundleResourceMap` closes the framework's global and localized resources.

Mark the Public Header Files

Public header files declare the public API for your framework. These are put inside your framework in a folder called Headers, and anyone who uses your framework has access to them.

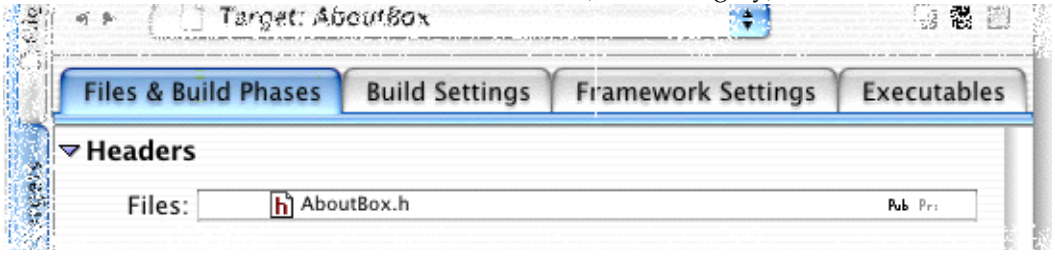
A framework can also have private and internal header files. Private headers are placed in your framework in a folder called Private Headers, and are usually removed from your framework when it's distributed to others. Internal headers are not placed in your framework.

The AboutBox framework has only one header file and it's public. In this step, you'll mark it as public.

1. Click the Targets tab, select AboutBox, and click the Files & Build Phases tab.
2. Turn on the public header option for `AboutBox.h`.

   If you can't see `AboutBox.h`, click the triangle beside Headers.

   To turn on the public header option, find the word "Pub" that's to the right of `AboutBox.h` and click it so it turns black (instead of gray).



If “Priv” is black, the header is private. If neither “Pub” nor “Priv” is black, the header is internal.

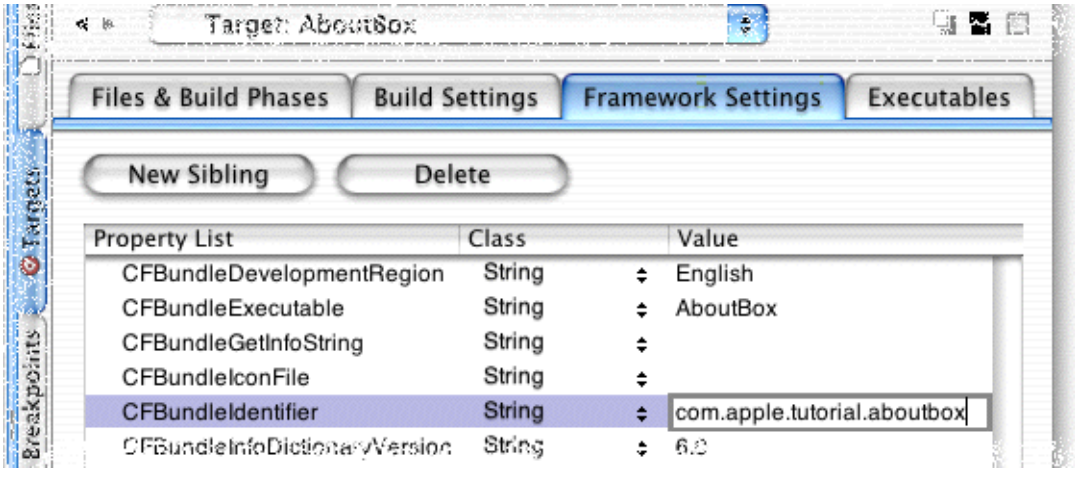
Keep the target’s editor open since you’ll use it in the next section.

Assign a Bundle Identifier and Executable Name to the Framework

The executable name is the name of the shared library file inside the framework. The bundle identifier is used by the framework’s code to find the bundle that contains its resources. To ensure that it’s unique, the bundle identifier should be Java-style package name; for example, `"com.mybusiness.myframework"` or `"edu.StateU.psych.myapp"`.

1. In the target editor for AboutBox, click the Framework Settings tab.
2. In the `CFBundleExecutable` field, enter `AboutBox`.
3. In the `CFBundleIdentifier` field, enter `"com.apple.tutorial.aboutbox"`.

   It should look like this:



   Note that this panel lists several other useful properties. For more information on what they do, see *Software Configuration* (at `/System/Documentation/Developer/SystemOverview/SoftwareConfig.pdf`).

Build the Framework

Click the column beside the AboutBox target so that a checkmark appears. This is the same as choosing a target from the pop-up menu above the files list.

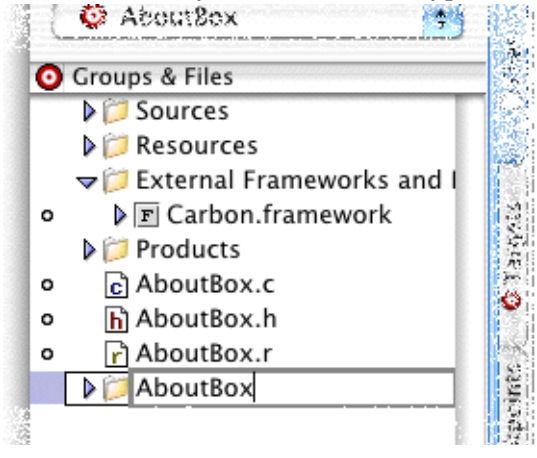
All the buttons along the top of the project window apply to the selected target. Click the Build button to build the framework.

Regroup the Files

Optionally, you can move the files into groups that make more sense: placing all the framework files into one group and all the application files into another. Here's one suggested way to do it.

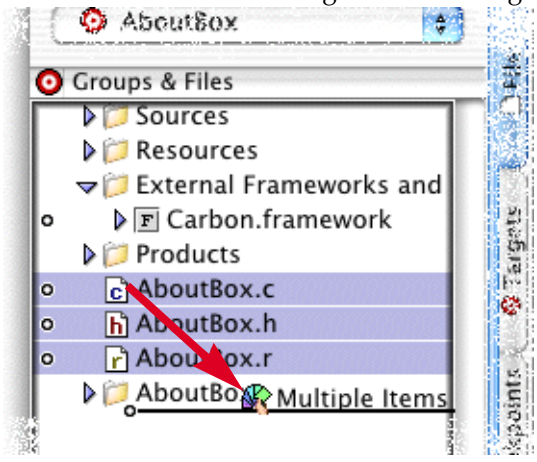
1. Create a new group and name it AboutBox.

   Choose Project > New Group. Project Builder creates a new group in the file list and automatically selects its name. Type `AboutBox` and press Return.



2. Move the files `AboutBox.c`, `AboutBox.h`, and `AboutBox.r` into the AboutBox group.

   Select all three files and drag them into the group.



3. Rename the Sources folder to AboutBoxApp.

   Select the Sources folder and choose Project > Rename. Project Builder selects its name. Type `AboutBoxApp` and press Return.

4. Move `main.r` and `InfoPlist.strings` from the Resources group into the AboutBoxApp group.

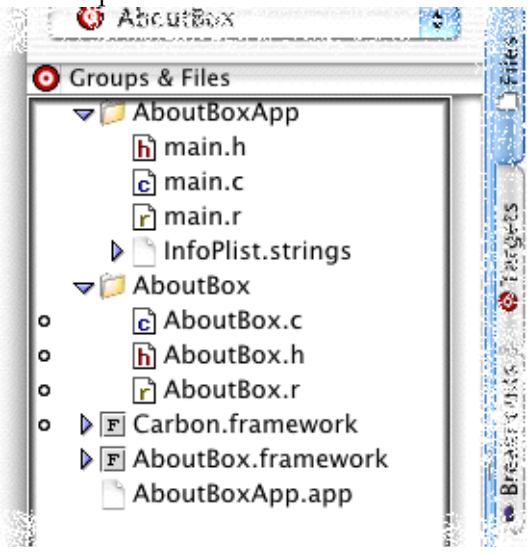
5. Move `Carbon.framework` out of External Frameworks & Libraries to the top level of the file pane.

6. Move `AboutBoxApp.app` and `AboutBox.framework` to the top level of the file pane.

7. Remove the empty groups: Resources, External Frameworks & Libraries, Products.

   Select the two groups and choose Edit > Delete. When Project Builder asks if you want to delete them from the disk as well, click No.

The file pane should look like this:



Even though `main.r` is no longer in the Resources group, Project Builder still treats it as a resource file. And even though you've changed the groups the files are in, you haven't changed where the files are on disk. If you go back to the Finder and look at the project's directory, you'll notice they're still there, in the same directory.

Add the Framework to the Test Application

Now you'll add the AboutBox framework to the project's AboutBoxApp target.

1. "Update the Test Application" (page 31)
2. "Replace the Application's Resource File" (page 31)
3. "Assign a Short Version String to the Application" (page 32)
4. "Add the Built Framework to the Project" (page 32)
5. "Make the Application Target Dependent on the Framework Target" (page 33)

Update the Test Application

In this sample, the test application is mostly written for you. All you need to do is include a header file and delete some items that are now in the framework.

1. In `main.c`, include `AboutBox.h`.

   Go to the beginning of `main.c`. After the `#include <MacWindows.h>` statement, add `#include "AboutBox.h"`.

2. In `main.c`, delete the declaration and definition of `DoAboutBox`.

   The declaration is soon after the include files and looks like this:

   ```
   void DoAboutBox(void);
   ```

   The definition is the last function in the file and looks like this:

   ```
   void DoAboutBox(void)
   {
       //Carbon currently has an event problem with modal dialogs
       //will put this back soon...

      //(void) Alert(kAboutBox, nil);  // simple alert dialog box
   }
   ```

3. In `main.h`, delete the definition of `kAboutBox`.

   It's the last line in the file and looks like this:

   ```
   #define kAboutBox200     /* Dialog resource for About box */
   ```

Replace the Application's Resource File

Right now, `main.r` contains resources for both the application and the framework. In this step, you'll replace that file with one that contains resources for only the application.

1. Remove `main.r`.

   Select `main.r` and choose Edit > Delete. When Project Builder asks whether to delete the file from the disk as well, press Delete.

2. Add the new `main.r` to the AboutBoxApp target.

   Choose Project > Add Files, and select `main.r`, which should be in the same folder as this tutorial (`/System/Documentation/Developer/DeveloperTools/PBX/`

AboutBox/). Select "Copy into group's folder," and make sure AboutBoxApp is checked and AboutBox is not checked.

Project Builder copies the file into your project's directory and adds it to your project's file list and to the AboutBoxApp target.

Assign a Short Version String to the Application

The short version string contains the application's name and version number. The framework displays it in the About box.

In InfoPlist.strings, change CFBundleShortVersionString to "AboutBoxApp 0.01d1". You can also change the CFBundleName and CFBundleGetInfoString if you like, but they're not used in this tutorial. The file should look like this:

Listing 2-2 InfoPlist.strings

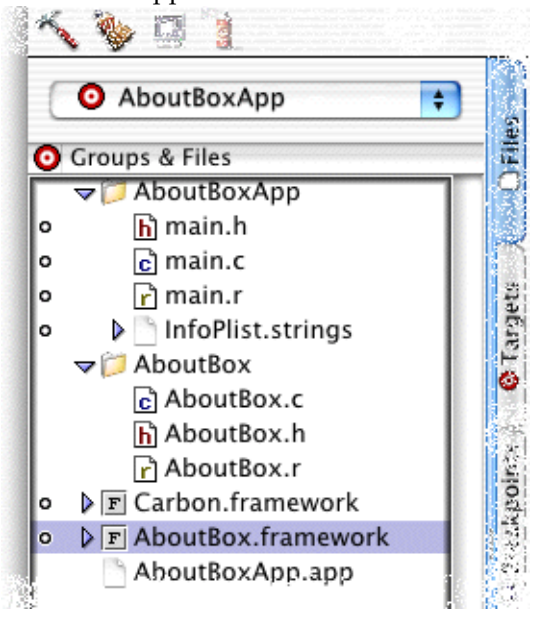
```
/* Localized versions of Info.plist keys */

CFBundleName = "AboutBoxApp";
CFBundleShortVersionString = "AboutBoxApp 0.01d1";
CFBundleGetInfoString = "AboutBoxApp version 0.0.1d1, Copyright 2000";
```

Add the Built Framework to the Project

You need to add the built framework to your application's target.

1. Select AboutBoxApp from the pop-up menu above the file list.
2. Click beside AboutBox.framework.

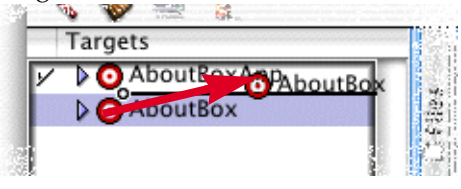
A circle appears beside it. It should look like this:



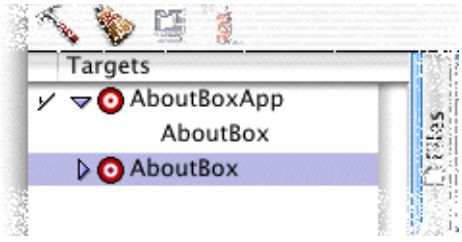
Make the Application Target Dependent on the Framework Target

Now you need to let Project Builder know that the application target is dependent upon the framework target. Say the framework's source files have changed since you last built it, and then you build the application target. As things stand now, Project Builder won't update the framework but will use the old version. After this step, Project Builder will rebuild the framework and use the rebuilt version.

Just click the Targets tab, and drag the AboutBox target onto the AboutBoxApp target.



If you click the triangle beside AboutBoxApp, you'll see AboutBox underneath it. That lets you know that AboutBoxApp now depends on AboutBox. If you build AboutBoxApp, it will make sure AboutBox is built before proceeding.



Build and Run the Test Application

Click the AboutBoxApp target's icon so an arrow appears in it. Then choose Build > Build and Run. Project Builder builds and runs your application.

In your application, choose AboutBox > About Hello, and look at your new About box.