INSIDE CARBON

Moon Travel Tutorial: Creating a Carbon Application

For Mac OS X

01/05/01

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

Simultaneously published in the United States and Canada.

Contents

Introduction: Moon Travel Tutorial

This tutorial shows how to use Project Builder and Interface Builder to create a simple Carbon application that calculates travel time to the moon and displays facts about the moon. The Moon Travel application is designed to run on Mac OS X and uses the Carbon Event Manager.

The Moon Travel tutorial covers a number of tasks, including the following:

- creating a new project in Project Builder
- using Interface Builder to create windows and menus
- writing and installing Carbon event handlers
- adding a PICT resource to the interface using Interface Builder
- creating a radio button group control and retrieving its setting
- displaying static text
- getting text from a localizable strings file
- displaying product and version information in an About window
- getting a window out of the Dock after it's been minimized
- adding information about the application to the application's bundle
- adding a Help book as an application resource
- launching the Help Viewer from the application's Help menu

Organization of the Tutorial

The tutorial is organized into seven chapters.

- "Basic Carbon Concepts" (page 11) provides a brief introduction to the terms used in the tutorial and lists the event type constants you'll use as you create the Moon Travel application. If you're already familiar with the Macintosh programming model, you'll find that several core concepts have changed for Mac OS X, such as event and resource handling.
- "Specifying the Application" (page 17) describes the goal of the Moon Travel application, and specifies what the user interface should contain and how it should behave. The specification should help you follow the steps for creating the interface more easily.
- "Using Project Builder to Create the Moon Travel Project" (page 21) shows how to create a Project Builder project from a template, and provides a brief introduction to Project Builder.
- "Using Interface Builder to Create Windows" (page 29) offers step-by-step instructions for creating the windows for the Moon Travel interface, attaching commands to buttons, and laying out an Aqua interface.
- "Using Interface Builder to Add Menu Items" (page 57) provides step-by-step instructions for adding a submenu to the Moon Travel interface and attaching commands to menu items.
- "Writing the Event Handlers and Other Code" (page 65) shows how to create the event and command handlers for the Moon Travel application.
- "Adding the Moon Travel Help Book" (page 85) describes how to add a help book to a Project Builder project, register the help book with the operating system, and open the help book to the title page.

Requirements

You need the following to complete this tutorial:

- a computer that's running Mac OS X
- Project Builder and Interface Builder, available on the Mac OS X Developer CD

CHAPTER 1

Introduction: Moon Travel Tutorial

Basic Carbon Concepts

This chapter covers the key Carbon-related concepts needed to write an application for Mac OS X using Carbon and the C programming language.

Carbon Events

Since the beginning of the Macintosh, most Macintosh applications have been event-driven—that is, applications respond to various changes or occurrences and take action based on the nature of the event. An **event** is the means by which the operating system communicates information about user actions, changes in the processing status of the application, hardware activity, and other occurrences that require a response from the application.

In the past, a typical Macintosh application would check repeatedly to see if an event occurred and, if so, respond to the event. If no events were pending, the application could choose to relinquish the processor for a specified amount of time or perform other tasks before checking again to see whether an event occurred. This sort of event checking and management is handled by the original (or classic) Event Manager. However, this style of event management is no longer recommended and is not used in this tutorial. Instead, you should use the handler-based event management supported by the Carbon Event Manager.

The Carbon Event Manager offers a simple yet flexible approach to event handling that greatly reduces the amount of code needed to write a basic application. It provides standard handlers for most types of user interaction, so you can concentrate on writing code that's unique to your application. You don't need to

write your own event handlers unless you want to override a default behavior. A single callback function, `InstallEventHandler`, is all you need to attach your own event handler to any Toolbox object.

The Carbon Event Manager's streamlined event handling enhances system performance on Mac OS X through more efficient allocation of processing time. Applications that use the Carbon Event Manager not only run better on Mac OS X, they help improve overall performance and responsiveness, creating a better experience for our customers.

Handler-based event management is fairly straightforward. It involves these steps:

1. Identify the events your application must handle.
2. Write handlers (functions) to respond to the events.
3. Register the handlers with the operating system. This informs the operating system of the events your application handles, and which handlers respond to which events. To register a handler, you simply call an "InstallHandler" function that passes a pointer to the handler along with information about the events associated with the handler.
4. Call the function `RunApplicationEventLoop` and the Carbon Event Manager does the rest. When the Carbon Event Manager detects an event that your application handles, the manager calls your handler.

Event Types

Every Carbon event is characterized by an **event type**, which consists of an event class and an event kind. The **event class** denotes a general category of events, such as window or mouse events. The **event kind** indicates a specific event within the category, such as a window-close event. A class may have many events associated with it. As you identify the events your application handles, you need to find the pair of constants (event class and event kind) associated with each event. You can find these constants in the Carbon Event Manager reference documentation or in the `CarbonEvents.h` header file.

The Moon Travel application uses the event classes and event kinds listed in Table 2-1. The event class `kEventClassCommand` denotes a menu or other command. The Moon Travel application processes commands from the main menu as well as those issued by buttons in the main window.

Table 2-1 Event classes and event kinds used in the Moon Travel application

Event Class	Event Kind
`kEventClassCommand`	`kEventProcessCommand`
`kEventClassWindow`	`kEventWindowClose`
`kEventClassWindow`	`kEventWindowActivated`
`kEventClassWindow`	`kEventWindowClickContentRgn`

Event References

When the Carbon Event Manager detects an event for your application, it returns an event reference to your application. The **event reference** is an opaque structure that contains general information about the event's class, kind, and time of occurrence. Because it is opaque, you must use one of the Carbon Event Manager's accessor functions to get the data associated with the event reference.

The specific function you use to access the event reference depends on the type of event. You use the function `GetEventKind` to access the kind of event (window activated, window close, and so forth); the function `GetEventClass` accesses the event class and the function `GetEventTime` returns the time the event occurred.

Event Parameters

An **event parameter** is an opaque structure that contains specific information associated with an event, including the parameter name and parameter type. More than one parameter can be associated with an event. A mouse-down event has four parameters that specify mouse location, button pressed, modifier keys, and number of mouse clicks.

You use the Carbon Event Manager function `GetEventParameter` to retrieve the specific data associated with an event parameter. You must supply the parameter name and parameter type of the data you want to retrieve. Parameter names and types are listed in the Carbon Event Manager reference documentation or in the `CarbonEvents.h` header file.

The Moon Travel application uses only one event parameter, specified by the parameter name `kEventParamDirectObject` and the parameter type `typeHiCommand`. You can use these values with the function `GetEventParameter` to retrieve the command ID of an event of class `kEventClassCommand`. Your application can then use the command ID to determine what action to take, such as opening a window or computing travel time to the moon.

Event Targets

An event handler is associated with an event target. An **event target** can be an object associated with the interface (such as a window, menu, or control) or it can be the application itself. The Moon Travel application uses only windows as event targets.

An event target can have any number of events associated with it. For example, a handler for an application's main window (the event target) could process command events, window events, and mouse events.

Default Event Handlers

If you are not doing anything out of the ordinary in your application, you can use one of the Carbon Event Manager's default event handlers. A default event handler provides a standard response to each type of event received by an event target. For example, the default window handler implements all the standard behavior for manipulating a window with the mouse—closing it when the user clicks the close button, resizing it when the using drags the resize control, and so on. You only need to create and install a handler to take care of those aspects of the window's behavior that are specific to your application.

If your handler gets an event that it does not handle, the Carbon Event Manager can use a default handler to process the event.

Resources

Resources are a basic element of every Macintosh application. Resources typically include data that describe menus, windows, controls, dialog boxes, sounds, fonts, and icons. Applications and system software interpret the data for a resource according to its resource type. Because such resources are separate from the application's code, you can easily create and manage resources for menu titles, dialogs, and other parts of your application without recompiling every time you change one of the resources. Resources simplify the process of translating interface elements containing text into other languages.

In earlier versions of the Mac OS, resources were stored in a file's resource fork. In Mac OS X, files do not have resource forks. Instead, resources are stored in files located in a resources directory. Although resources can be created in a number of ways, including with tools such as Rez, the Moon Travel application uses Interface Builder to create its interface-related resources.

Interface Builder is a WYSIWYG application from Apple Computer that lets you quickly create interface elements using graphical tools. One of Interface Builder's advantages is that it supports the Aqua interface layout guidelines, which can save you a lot of work. Once you create and save your interface, Interface Builder generates a "nib" file that contains interface-related resources. (Nib is an acronym for "Next Interface Builder.") Then, from within your application you call various functions, such as the functions `CreateWindowFromNib` and `SetMenuBarFromNib`, to create the interface elements from the nib file.

It's also possible to use old-style resources in Mac OS X. The Moon Travel program uses a PICT resource in the interface. The PICT is stored in a `.rsrc` file located in the resources directory.

CHAPTER 2

Basic Carbon Concepts

Specifying the Application

Before you use Project Builder and Interface Builder to create the Moon Travel application, read the goal definition and interface specification for Moon Travel in this chapter. They should help provide you with an understanding for creating the Moon Travel application.

Defining the Goal for Moon Travel

Most people have difficulty understanding astronomical distances. Astronomers use measurements, such as light years, but it's often easier to comprehend the vastness of planetary and stellar distances by relating them to something we have experience with, such as walking, driving, or flying. The Moon Travel application addresses this problem by translating distance into the number of days it would take to travel to the moon, given a particular mode of transportation specified by the user.

Specifying the Interface

Before you use Interface Builder to create an interface, list the interface elements you'll want and have a rough idea of how you'd like them to appear and to behave. The Moon Travel application has a main menu and three windows (main, Moon Facts, and About).

The Main Window

The main window displays a photograph of the moon, a list of modes of transportation, a field that shows computed travel time, and three buttons. One button computes travel time, another opens a window that displays facts about the moon, and the third quits the application. The window's title bar displays the words "Moon Travel Time."

The window opens when the application launches and closes when the application quits. The user can minimize the window so its icon displays in the Dock. The user can quit the program by clicking a Quit button or by choosing Quit from the File menu.

The window is moveable but not resizable, and it becomes inactive when the Moon Facts window opens.

The Moon Facts Window

The Moon Facts window displays static text containing information about the moon. The window's title bar displays the words "Moon Facts."

The window opens when the user presses a button in the main window or chooses the moon Facts command from the moon menu. The user can close or minimize the window.

The user can't modify or copy the text displayed in the window, or resize the window. The window is moveable and becomes inactive when the user makes the main window active.

The About Window

The About window displays the Moon Travel application icon and version and copyright information about the Moon Travel application. The window's title bar displays the words "About Moon Travel."

The window opens when the user chooses About Moon Travel from the Moon Travel App menu. The user can close the window.

The user is not able to modify or copy the text or change the size of the window.

The Main Menu

In addition to standard menu items, the main menu has a Moon menu with two commands—one to open the Moon Facts window and another to compute travel time. The About menu item opens the About window, and the Help menu item opens the Moon Travel Help to its table of contents page.

CHAPTER 3

Specifying the Application

Using Project Builder to Create the Moon Travel Project

This chapter shows you how to create a project with Project Builder, Apple's integrated development environment (IDE) for Mac OS X. Project Builder provides project editing, searching, navigation, file editing, project building, and debugging for such Mac OS X software projects as applications, tools, frameworks, libraries, plug-in bundles, kernel extensions, and device drivers.

You'll create the Moon Travel project from a template, then build and run it. In later chapters you'll create the interface and write the code for the Moon Travel application.

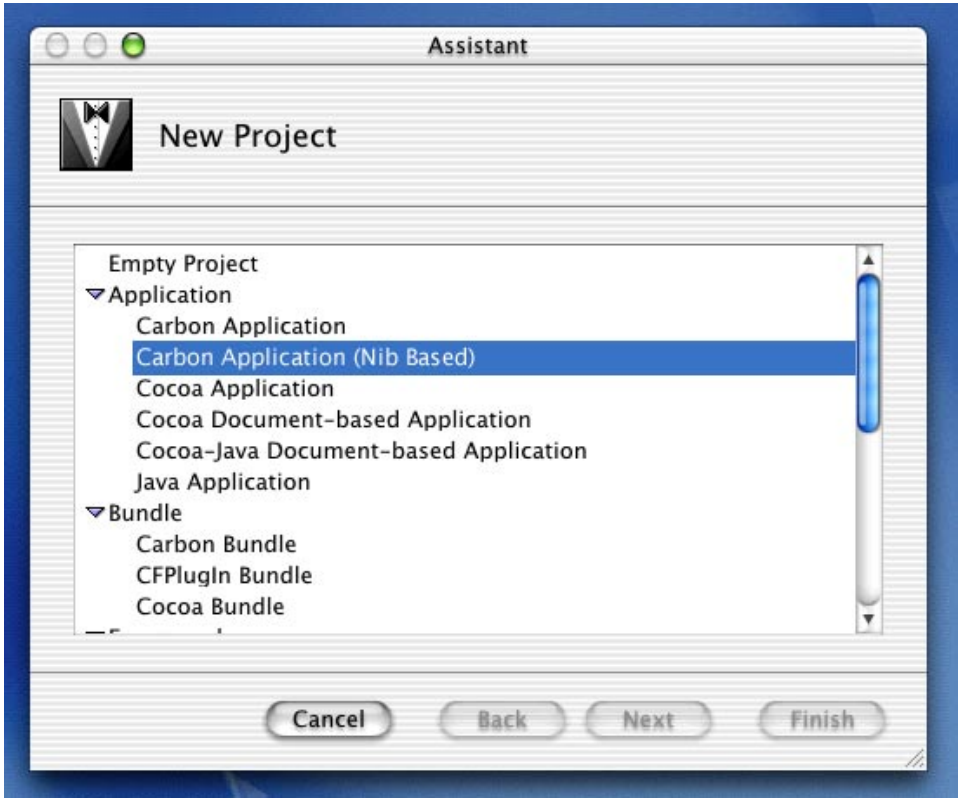
Creating the Project

To create the Moon Travel project, do the following:

1. Double-click the Project Builder icon located in Mac OS X in the folder `/Developer/Applications`.

2. Choose New Project from the File menu.

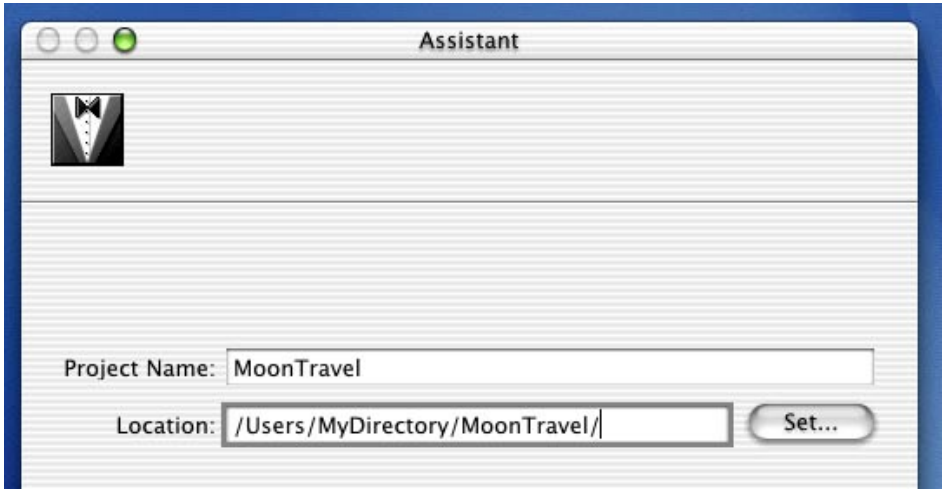
A dialog opens with a list of template objects.



3. Select Carbon Application (Nib Based), then click Next.

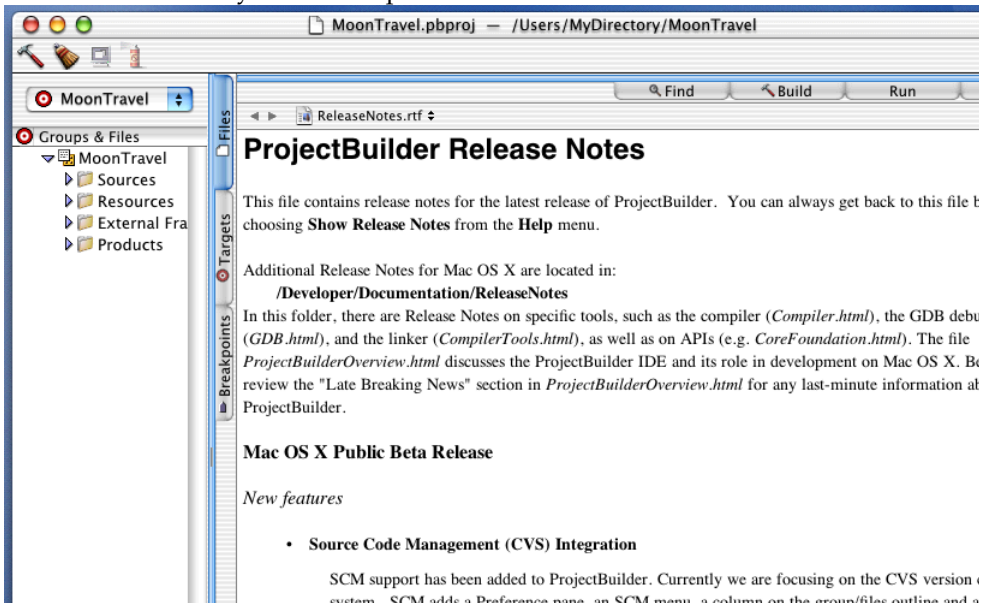
   Nib refers to a project that uses Interface Builder to create the interface. You'll use Interface Builder later.

4. Enter `MoonTravel` as the project name and click Set to choose a location to store the project.



5. Click Finish.

   Project Builder creates a directory for you, places a project file and some source files in it, and opens the project's window. The project already contains sample source files that you can compile and run.



Take some time to look at what's in the Moon Travel project. If you already understand what projects, targets, and frameworks are, you can move ahead to "Build and Run the Sample Application" (page 27).

Project Builder Items and Groups

A Project Builder project contains two types of items: file references and targets. The files are in a list at the left of the project window and the targets are in the pop-up menu above the files list. The MoonTravel target is shown in the pop-up menu in Figure 4-1 (page 25), with files arranged in groups in the list below the target pop-up menu.

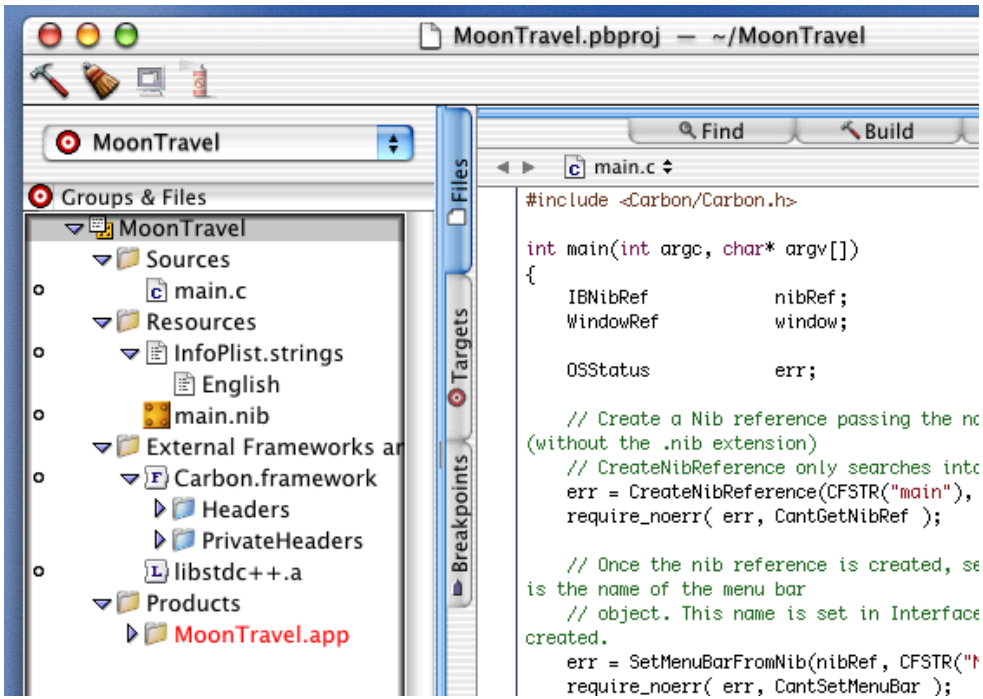
- Files can be references to source files, resource files, libraries, and frameworks. The files themselves aren't in your project. You can place related files together in groups. If a circle is beside a file, it's used by the target selected in the Target menu.
- Targets are products you can build from your project's files. A simple project, such as the Moon Travel project, has just one target that builds an application. A complex project may contain several. For example, a project for a client-server software package could contain targets for
  - a client application
  - a server application
  - a private framework that both applications use
  - command line tools that you can use instead of the applications

  The tutorial "AboutBox: Creating a Framework With Project Builder" shows how to manage projects with multiple targets. You can download the tutorial from the Mac OS X Developer Documentation website (see Developer Tools):

  http://developer.apple.com/techpubs/macosx/

Project Builder organizes the files it created for the Moon Travel project into four groups: Sources, Resources, External Frameworks and Libraries, and Products. To see what 's in a folder, click the disclosure triangle next to its name.

Figure 4-1 MoonTravel target with files arranged in groups

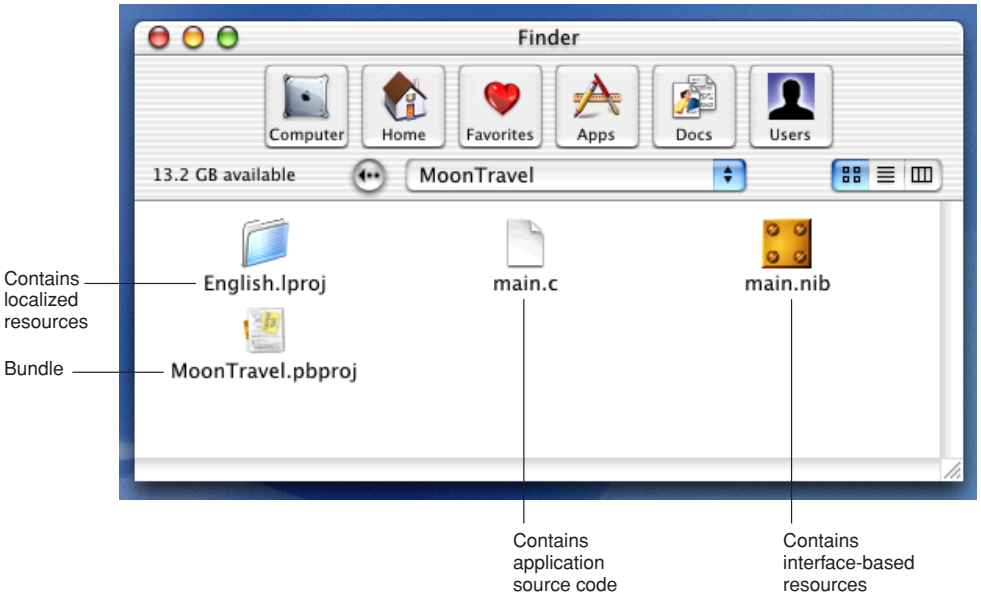


- Sources: These are files that contain the application's source code and are compiled to produce object code. In Project Builder, header files are listed in the project window and are in this group. This project contains only one source file: `main.c`.
- Resources: These are files that contain resources or that can be compiled to produce resources. This project contains two: `main.nib`, which contains the application's interface-based resources, and `InfoPlist.strings`, which contains strings that can be localized. You'll read more about these later.
- External Frameworks and Libraries: This project contains a framework and a library. In Mac OS X, a framework is a shared library that's bundled with its header files and resources. Click the disclosure triangle next to `Carbon.framework` to see a list of its header files.
- Products: These are the items the targets in your project produce. Right now, there is one product, `MoonTravel.app`.

You can move the files into any groups you want. The groups are there solely for your convenience and do not affect Project Builder's ability to find or compile the files.

Now go back to the Finder and take a look at the files in your project's folder. They look a bit different from the view you see in Project Builder.

Figure 4-2 Finder view of Moon Travel project files



`MoonTravel.pbproj` is a **bundle** that keeps track of your project's files and targets. (A bundle is a directory in which executable code is stored along with related resources. It's a folder of files that the Finder treats as a single unit.) If you want to see its contents, you can Control-click the `MoonTravel.pbproj` icon.

`English.lproj` contains resources that are localized into English. In this case, it contains `InfoPlist.strings`. You can have `lproj` folders for other languages, in which case the folder is named for the language, such as `French.lproj` or `Japanese.lproj`.

Notice that although `main.nib` is in a different Project Builder group from `main.c`, the two files are in the same folder. Also notice that `Carbon.framework` isn't in the project directory. It's in `/System/Library/Frameworks`. The project contains a reference to it.

Build and Run the Sample Application

Build and run the sample application to get an idea of how Project Builder works. The sample code should run without modification.

1. Click the Build button in the upper-left corner of the project window.

   The Build panel appears. When the build finishes, a message appears at the bottom of the project window. As long as you have not changed the source files, you should not see any error messages.

   To hide the Build panel, click the Build tab.

2. Click the Run button in the upper-left corner of the project window.

   Project Builder launches the application and the Run window appears. At this point, the application displays an empty window.

   The Run window displays messages written to `stdout` and `stderr`. You can perform simple debugging by writing status information to these streams. You perform more sophisticated debugging with Project Builder's debugger, as shown in the tutorial *DebugApp: Debugging and Application With Project Builder*, available through the Mac OS X Developer Documentation website (see Developer Tools):

   http://developer.apple.com/techpubs/macosx/

3. Press Command-Q to quit the sample application.

Using Interface Builder to Create Windows

In this chapter you'll use Interface Builder to create the application windows as they are described in "Specifying the Interface" (page 17).

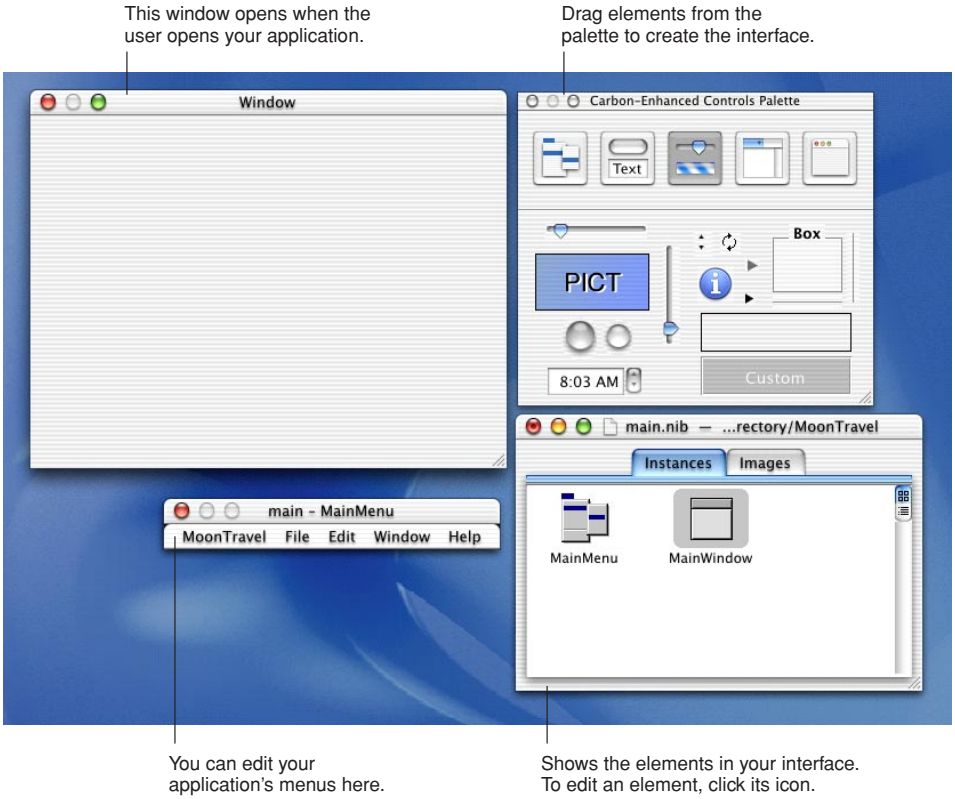
1. "Open the Nib File" (page 29)
2. "Create the Interface for the Main Window" (page 31)
3. "Create the Moon Facts Window" (page 47)
4. "Create the About Window" (page 50)

Open the Nib File

A **nib file** stores your application's resources. A nib file is an XML text file that describes your application's windows, menus, buttons, text fields, and other user interface elements.

1. Open the Moon Travel project if it is not already open.
2. Double-click the `main.nib` file, located in the Resources group of the Moon Travel project files.

Interface Builder opens these four windows:



- Window: Displays when your application runs. An application can use any number of windows
- Carbon Palette: Contains controls, menu items, and windows that you can drag into your interface. The Carbon Palette has five panes; the Enhanced Controls pane is shown above.
- Main Menu: Lets you edit the items that appear in your application's menu bar. It already contains common commands such as About, Quit, Save, Close, Copy, and Paste. May of the menu items (Quit, for example) behave correctly without your needing to write any code.
- main.nib: Displays the top-level items in your nib file.

Create the Interface for the Main Window

In this section, you'll create the interface for Moon Travel application's Main window.

1. "Add Items to the Main Window" (page 31)
2. "Add a Picture of the Moon to the Project" (page 41)
3. "Set Attributes for the Main Window" (page 42)
4. "Align Objects" (page 44)

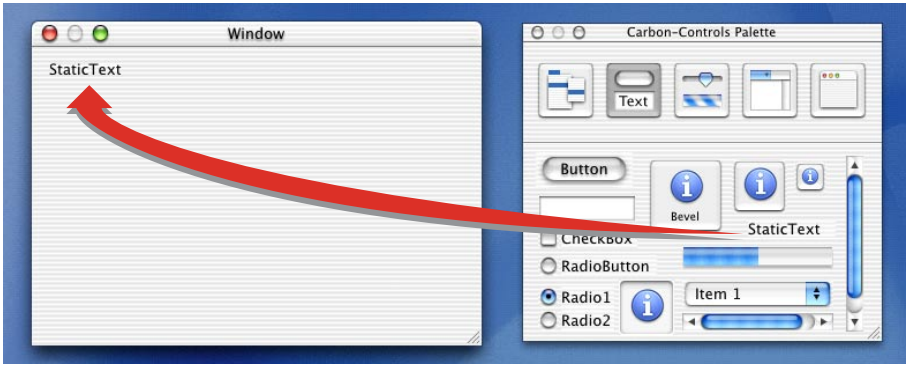
Add Items to the Main Window

You need to add several items to the Main window: two static text objects, a text field, a radio button group, three buttons, and a PICT of the moon.

1. Click on the Text button in the Carbon palette, if necessary.
2. Add static text to the top-left corner of the window.

   This is the label for the radio button group you'll add later.

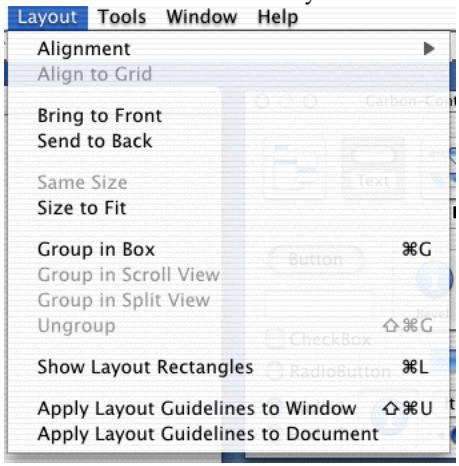
   From the palette, drag the object named Static Text to the upper left corner of the window.



3. Enter `Mode of Transportation` as the static text field's value.

Double-click the static text field, type `Mode of Transportation`, and press Return.

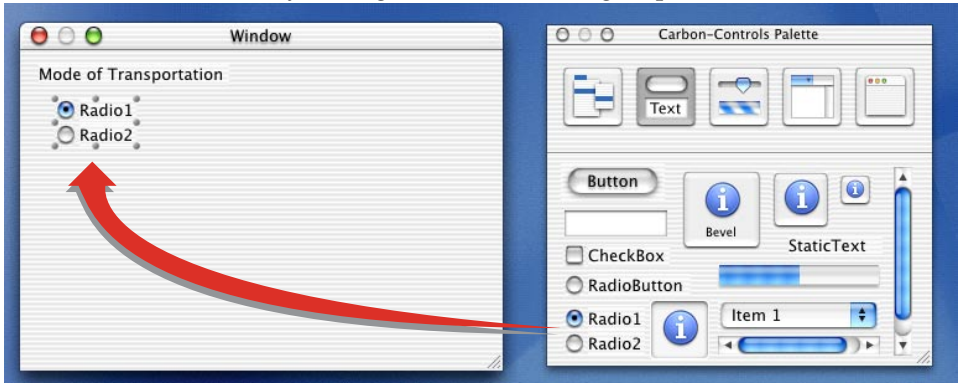
If the field is too small and part of the text is cut off, select the field and choose Size to Fit from the Layout menu.



4. Add a radio button group below the Mode of Transportation static text field.

   The user will select a mode of transportation from this group of radio buttons.

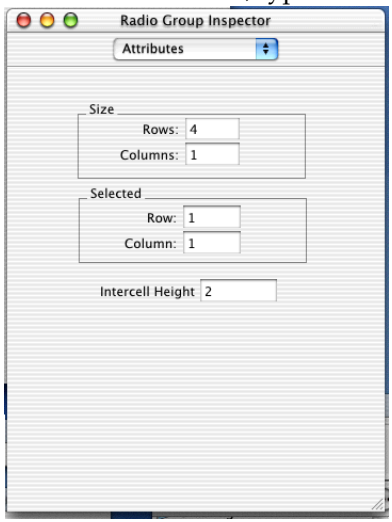
   From the palette, drag the radio button labeled Radio 1 to the area just below the Mode of Transportation text field. The Radio 1 and Radio 2 radio buttons move as a unit because they belong to a radio button group.



5. Increase the number of radio buttons in the radio button group to 4.

   With the radio button group selected, choose Inspector from the Tools menu. Choose Attributes from the Radio Group Inspector pop-up menu.

In the Rows field, type `4` in the then close the Radio Group Inspector window.



6. Resize the radio button group by dragging its corner so you can see all four radio buttons.
7. Label the radio buttons: Foot, Car, Commercial Jet, and Apollo Spacecraft.

   Double-click Radio 1 so the text becomes editable, type `Foot` and press Return. Use this procedure to label the other three radio buttons.

   Resize the radio button group by dragging its corner so you can see all four labels.
8. Enter `trav` as the radio button groups signature and `130` as its ID.

   In the Radio Group Inspector, choose Control from the pop-up menu. In the Control ID box, type `trav` in the Signature field and `130` in the ID field. You need these values later when your program reads the radio button group setting.
9. Add a static text label to the area below the radio button group.

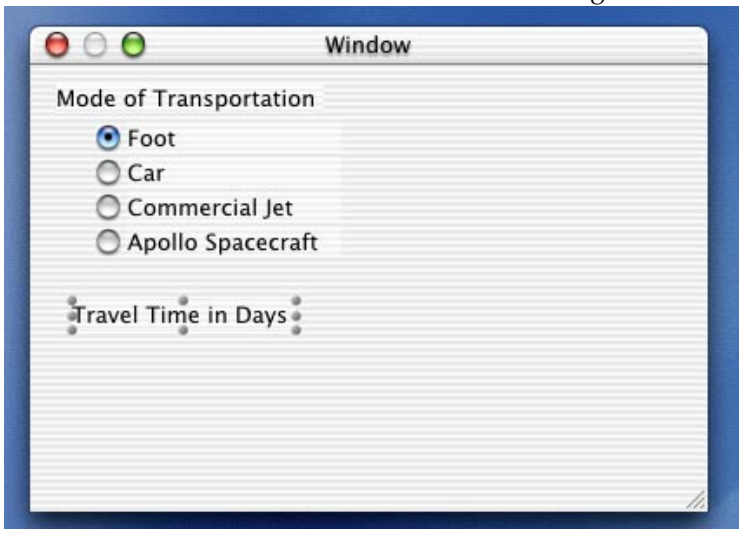
   This labels the field that displays the computed travel time to the moon.

   From the palette, drag the object named Static Text to the area below the radio button group.
10. Enter `Travel Time in Days` as the static text field's value.

    Double-click the field, type `Travel Time in Days`, and press Return.

    If the field is too small for the text, choose Size to Fit from the Layout menu.

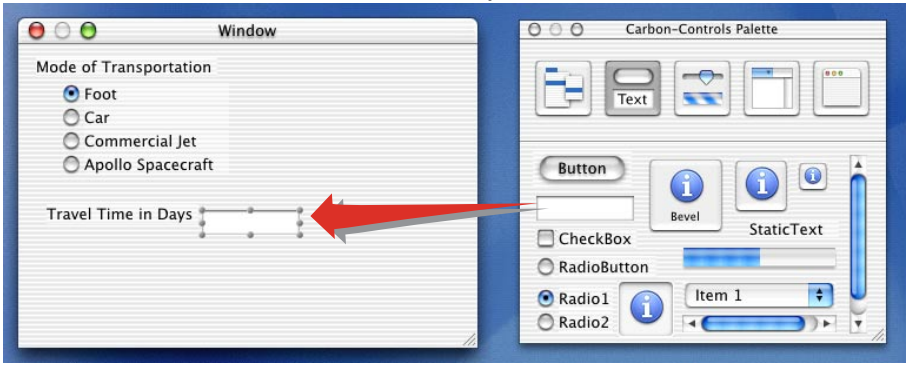
The main window should now look something like the following:



11. Add a text field to the right of the Travel Time in Days label.

    This is the field in which the computed travel time is displayed.

    Drag the white box under the push button labeled "Button" from the palette to the area next to the Travel Time in Days label.



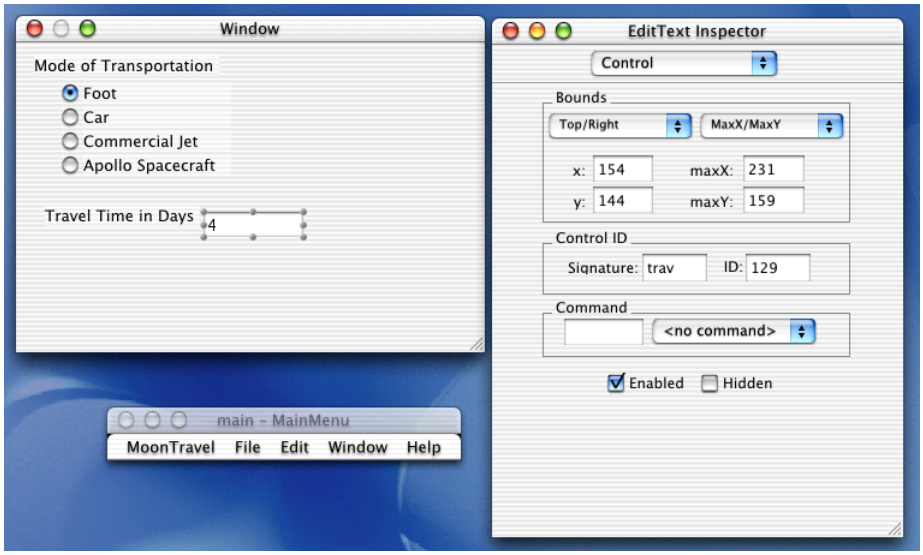
12. Enter `4` as the text field's value.

    Double-click the field, type `4` and press Return.

13. Enter `trav` as the text field's signature and `129` and the text field's ID.

    With the text field selected, choose Inspector from the Tools menu.

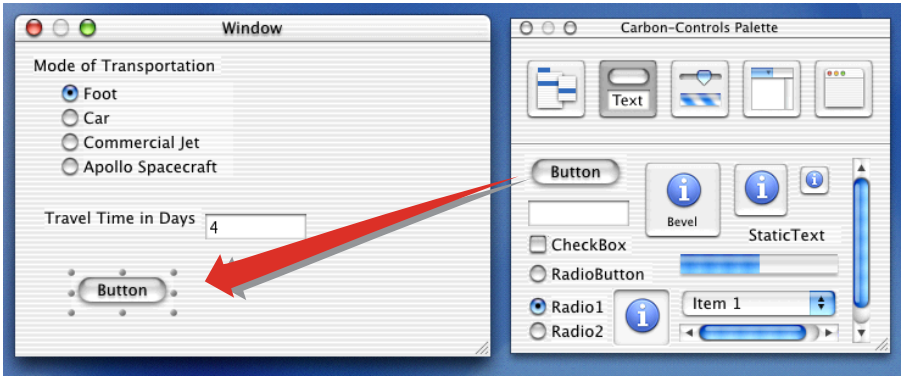
Choose Control from the pop-up menu in the Inspector window. In the Control ID section, type `trav` in the Signature field and `129` in the ID field. You need these values later when your program writes the computed travel time to the text field.



14. Add the Compute Travel Time button.

    The Compute Travel Time button issues a command that calculates the travel time to the moon in days based on the selected mode of transportation.

    Drag the push button from the palette to the area below the "Travel Time in Days" label.



15. Enter `Compute Travel Time` as the button's text.

Double-click the button, type `Compute Travel Time`, and press Return.

If the text isn't visible, select the button and choose Size to Fit from the Layout menu.

16. Enter `trav` as the button's command.

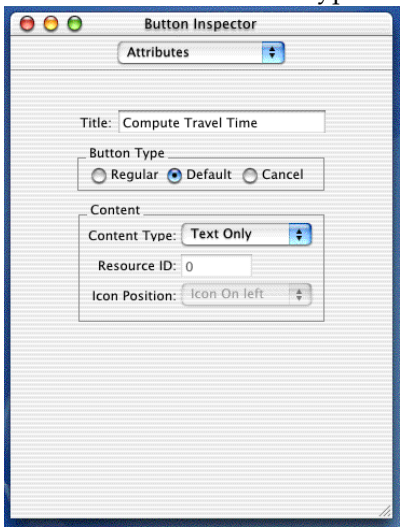
    Choose Control from the pop-up menu in the Button Inspector, type `trav` in the Command text field, and press Return.

    When the user presses this button, it sends a `trav` command to the Carbon Event Manager, which in turn calls your handler for the `trav` command. You'll define the handler in the chapter "Writing the Event Handlers and Other Code" (page 65).

17. Make the Compute Travel Time button the default button.

    A default button pulses and is the button that's selected when the user presses Return.

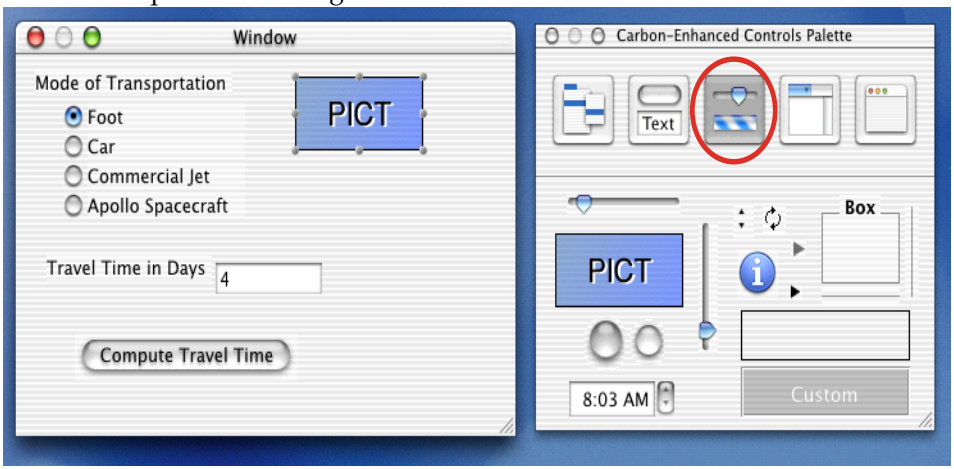
    Choose Attributes from the pop-up menu in the Button Inspector and select Default as the Button Type.



18. Add a PICT placeholder to the top, right-side of the interface.

    Later, you'll replace the PICT placeholder with a picture of the moon.

In the Carbon Palette, click the Enhanced Control button, then drag the PICT box from the palette to the right side of the window.



19. Resize the PICT box so it is 100 pixels square.

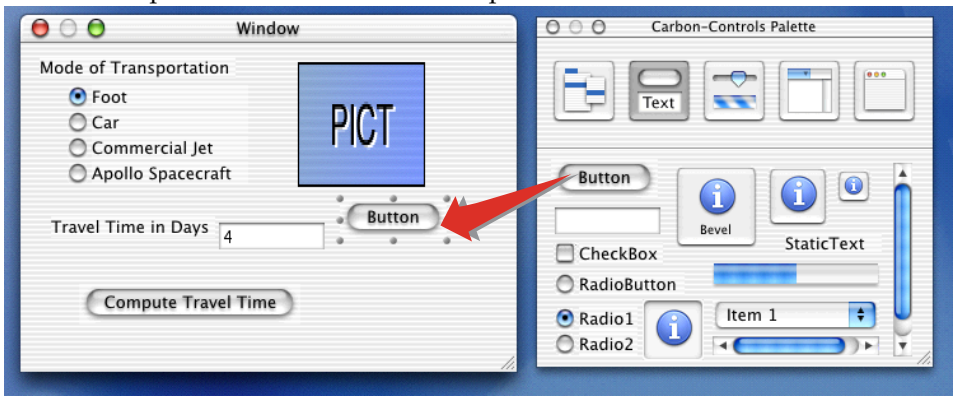
    This size matches the size of the moon picture supplied with this tutorial. When you add the picture of the moon later, the picture will assume the size of the PICT box.

    With the PICT selected, choose Inspector from the Tools menu, choose Control from the pop-up menu, and type 100 in the height and width fields.

20. Add the Moon Facts button.

    The Moon Facts button will open a window that displays facts about the moon.

Click the Controls button in the palette, then drag the button labeled Button from the palette to the area below the picture of the moon in the Main window.



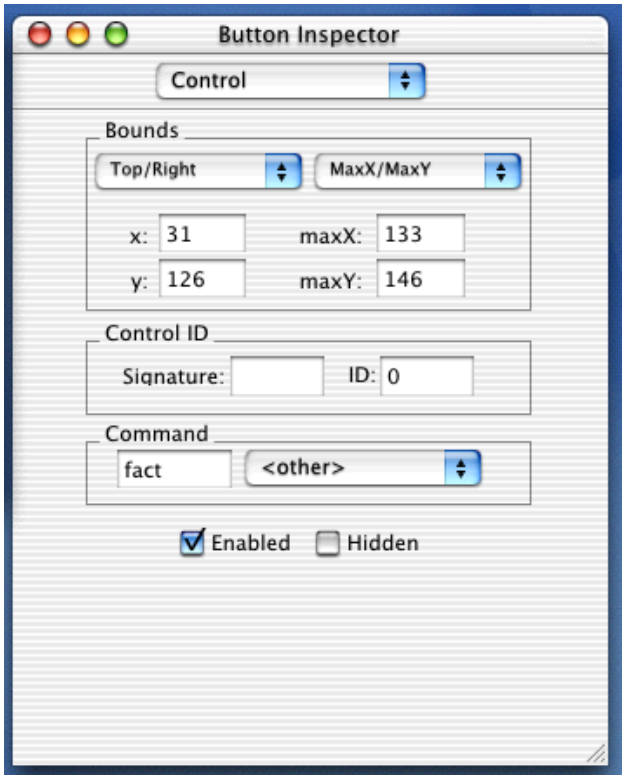
21. Enter `Moon Facts` as the button's text.

    Double-click the button, type `Moon Facts`, and press Return.

    If the text isn't visible, choose Size to Fit from the Layout menu.

22. Enter `fact` as the button's command.

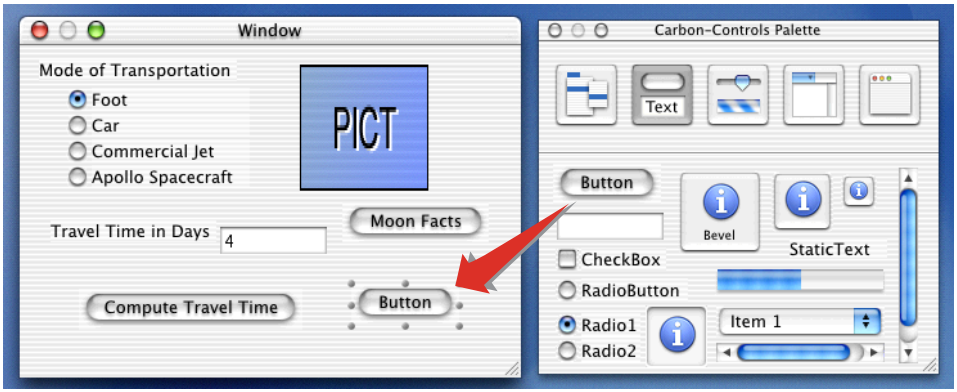
Choose Control from the pop-up menu in the Button Inspector and type `fact` in the Command text field.



When the user presses this button, it sends a `fact` command to the Carbon Event Manager, which in turn calls your handler for the `fact` command. You'll define the handler later on in this tutorial.

23. Add a Quit button.

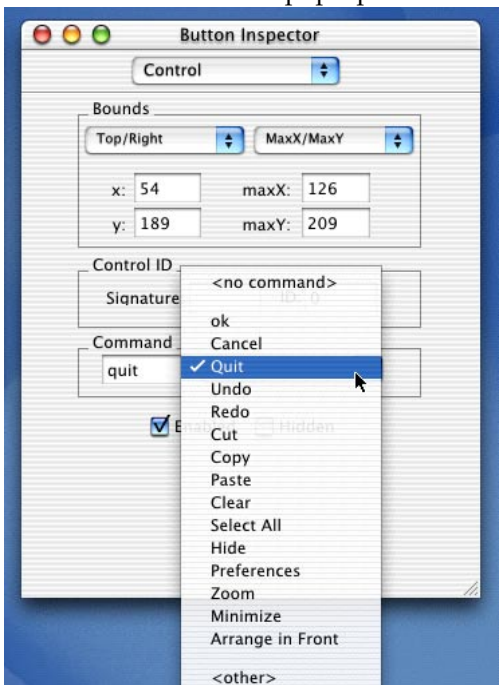
Drag a push button from the palette to the bottom, right side of the main window.



24. Enter `Quit` as the button's text.

25. Enter `quit` as the button's command.

Choose Control from the pop-up menu in the Button Inspector, then choose Quit from the Command pop-up menu.



When the user presses this button, the button sends a `quit` command to the Carbon Event Manager, which in turn calls a built-in handler that quits the application.

26. Quit Interface Builder.

    You'll use Interface Builder again later. You need to quit from it now and open it later so the picture you add in the next section will show up in Interface Builder.

Add a Picture of the Moon to the Project

In this section, you'll add to the project a PICT resource which you'll use to replace the PICT placeholder in the interface.

1. Copy the `moon.rsrc` file to the same MoonTravel folder.

   You should use the `moon.rsrc` file supplied with this tutorial; it's a PICT resource. (You can create your own PICT resource by opening a picture using a graphics application, then saving the picture as a PICT resource, with the file extension `.rsrc`.)

2. Add the`moon.rsrc` file to the Moon Travel project.

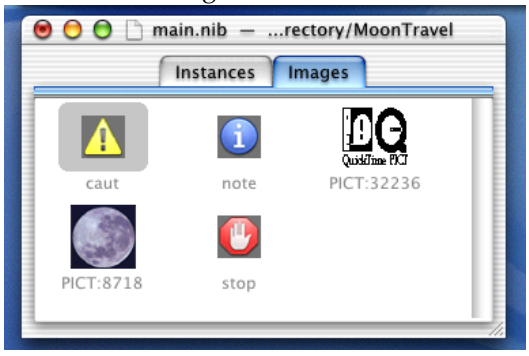
   Make the Moon Travel project active by clicking the Project Builder icon in the Dock. Then choose Add Files from the Project menu and select the `moon.rsrc` file.

   Click Open, select "Copy into group's folder," and click Add.

   Project Builder adds the file to the Resource group based on the file's extension.

3. Make Interface Builder active by clicking `main.nib` in the Project Builder file list.

4. Click the Images tab in the main.nib window.



If you don't see the moon PICT in the Images pane, quit Interface Builder, then open your `main.nib` file from the Project Builder again.

5. Drag the icon that looks like a photo of the moon to the PICT box.

The moon appears in place of the PICT, at the size of the PICT box. Your window should look similar to this. Later you'll align the items in the interface to make the window look more attractive.

Set Attributes for the Main Window

In this section you'll resize the window and change its name and type.

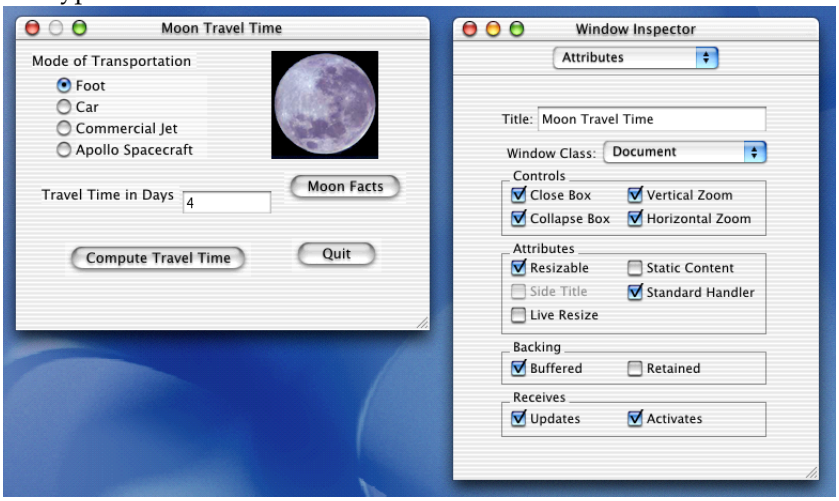
1. Resize the Moon Travel Main window.

   Click and drag the window's resize control so the window is just big enough to hold its items.

2. Change the window's name to Moon Travel Time.

   With the window active, choose Inspector from the Tools menu, then choose Attributes from the pop-up menu.

   Type `Moon Travel Time` in the Title text field.



3. Choose Document from the Window Class pop-up menu.
4. Set the window's controls.

   In the Controls group, make sure Collapse Box is the only control checked.

   Collapsing lets the user put the window in the Dock.

5. Set the window's attributes.

   In the Attributes group, make sure Standard Handler is the only control selected. This assures that any window activity for which you do not write a handler (for example, dragging the window) gets taken care of by the operating system.

You don't need to set the Backing or Receives options; just use the defaults.

Align Objects

Interface Builder has layout rectangles and other tools to help you align objects. Aligning interface objects is complicated by the fact the objects in Mac OS X have shadows which most user interface metrics don't take into account. Layout rectangles take shadows into account, so you should align the elements in your interface based on the rectangles rather than the objects themselves. You can view the layout rectangles of objects in Interface Builder by choosing Show Layout Rectangles from the Layout menu.

To set size values by hand or view the size of an object, you can use the Size pane of the Window Inspector (to set window size) and the Control pane of an object's inspector (such as the Button Inspector or the Radio Group Inspector).

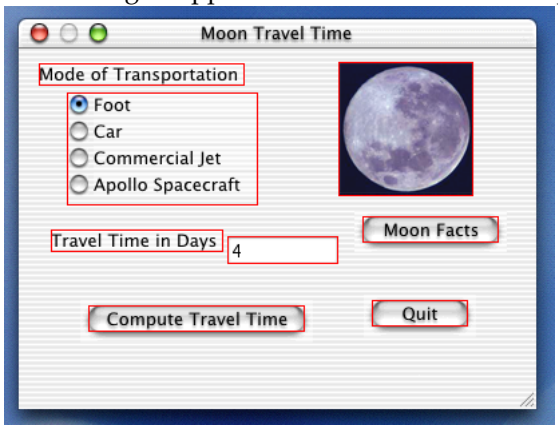
You can align objects in several ways using Interface Builder.

- Dragging objects with the mouse
- Pressing arrow keys (with the layout grid off, a selected object moves one pixel per key press)
- Using a reference object to put selected objects in rows and columns
- Using the built-in alignment functions
- Specifying origin points in the Size pane of the Inspector window
- Using a layout grid. (You can turn the layout grid on or off by choosing Grid > Turn Grid On or Grid > Turn Grid Off from the Tools menu.)

Use the built-in alignment functions by doing the following:

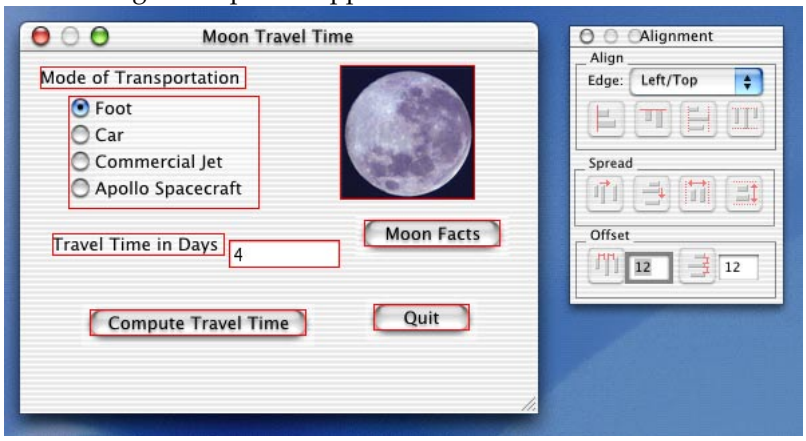
1. Choose Show Layout Rectangles from the Layout menu.

Rectangles appear around each interface object.



2. Choose Alignment from the Tools menu.

   The Alignment palette appears.



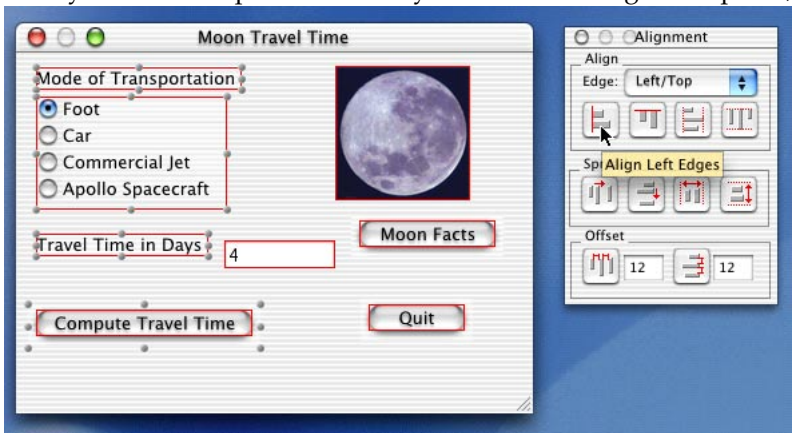
3. Align the items on the left side of the window.

   Select the Mode of Transportation text label, the radio button group, the Travel Time in Days text label, and the Compute Travel Time button by dragging a selection box around them or by pressing and holding the Shift key as you click each item.

   Choose Left/Top from the Edge pop-up menu in the Alignment palette.

   Click the Align Left Edges button in the top row of the Alignment palette.

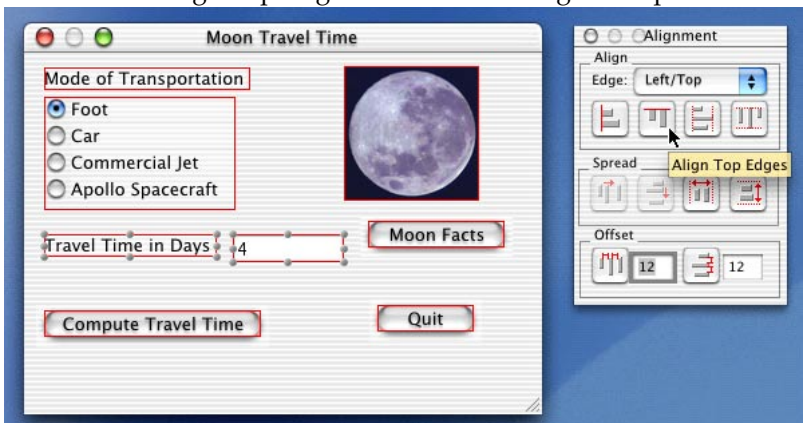
If you hover the pointer over any button in the Alignment panel, a label appears.



4. Align the Travel Time in Days text field with the Travel Time in Days label.

   Select the Travel Time in Days text field and the Travel Time in Days label.

   Click the Align Top Edges button in the Alignment palette.



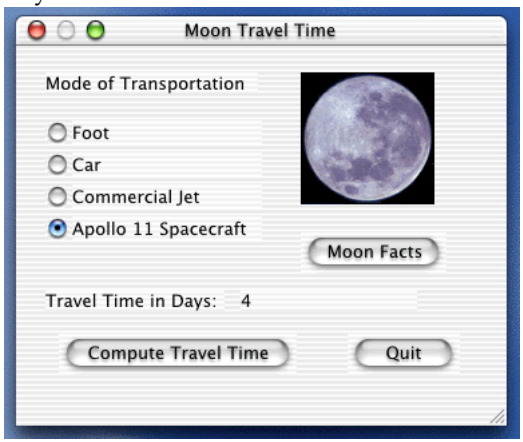
5. Align the top edges of the Mode of Transportation text label and the picture of the moon.
6. Align the right edges of the picture of the moon and the Moon Facts button.

   Choose Right/Bottom from the Edge pop-up menu in the Alignment palette.

   Click the Align Right Edges button in the Alignment palette.
7. Align the top edges of the Compute Travel Time button and the Quit button.

8. Choose Hide Layout Rectangles from the Layout menu.

By now, the Moon Travel Time window should look similar to the following. If it doesn't you can use the Alignment palette and the Size and Control panes of the Inspector to achieve the desired layout. You might also experiment with the "Apply Layout Guidelines to Window" command in the Layout menu.



Create the Moon Facts Window

In this section, you'll create a Moon Facts window that displays a few facts about the moon. The facts consist of a text string stored in a `Localizable.strings` file you'll create.

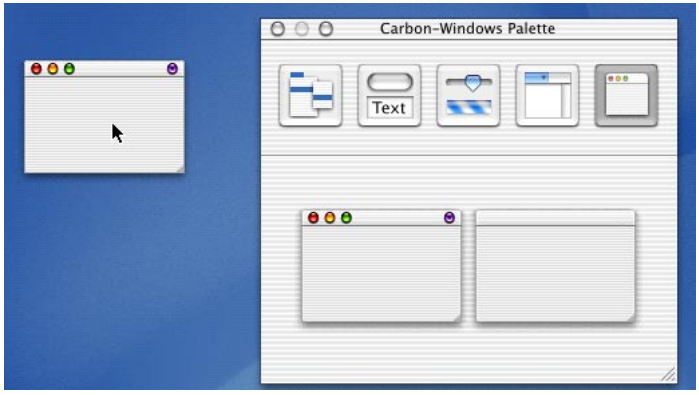
1. "Create a Window to Display Moon Facts" (page 47)
2. "Create Content for the Moon Facts Window" (page 49)

Create a Window to Display Moon Facts

1. Create a window.

   This is the window you'll use to display facts about the moon.

Click the Windows button in the Carbon palette. Then drag the icon of the window as shown.



2. Name the window `Moon Facts`.

   In the Instances pane of the main.nib window, double-click the word "Window" that's under the icon of the window you just created, type `Moon Facts`, and press Return.

3. Name the title bar of the window `Moon Facts`.

   With the window active, choose Inspector from the Tools menu, choose Attributes from the Window Inspector pop-up menu, type `Moon Facts` in the Title text field, and press Return.

4. Choose Document from the Window Class pop-up menu.

5. Set the window's controls.

   In the Control group, make sure Close Box and Collapse Box are the only options selected.

6. Set the window's attributes.

   In the Attributes group, make sure Static Content and Standard Handler are the only options selected.

7. Resize the window to 325 by 100 pixels.

   Choose Size from the Window Inspector pop-up menu, choose Width/Height from the right Content Rect pop-up menu, and type 325 for width and 100 for height.

8. Position the Moon Facts window where you'd like it to appear when the user opens it.

Where you position the window in Interface Builder determines its position when the user opens it. You should make sure that when the Moon Facts window opens the user can also see the main window.

9. Choose Save All from the File menu.

   Next you'll use Project Builder to add the window's content, so you should make sure your Interface Builder work is saved. But don't close Interface Builder, as you still need to create another window and set up the Main menu.

Create Content for the Moon Facts Window

You need to add the text you want displayed in the window. Although you could use Interface Builder to type the text directly into the Moon Facts window, for localizability it's best to store the text in a `Localizable.strings` file in the Moon Travel project. Later you'll write code that reads the `Localizable.strings` file and writes the string to the Moon Facts window.

1. Make the Moon Travel project window active.

   Click the Project Builder icon in the Dock.

2. Add a `Localizable.strings` file to your project.

   Choose New File from the File menu, create an empty file named `Localizable.strings`, and add it to the Resources group in your project.

3. Open the `Localizable.strings` file.

   Click `Localizable.strings` in the Groups & Files list. The empty file opens in the right side of the Project Builder window.

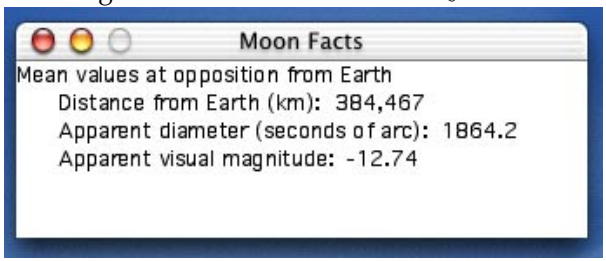
4. Add a text key and the text to the file.

   Type the following into the `Localizable.strings` file:

```
{
"Facts" = "Mean values at opposition from Earth \r     Distance from Earth
(km):  384,467 \r      Apparent diameter (seconds of arc):  1864.2 \r
Apparent visual magnitude:  -12.74"
}
```

The file must begin and end with curly brackets. The contents must consist of a text key (`'Facts'`) followed by an equal sign followed by the text associated with the key. Both the key and the string must be enclosed by straight quotation marks. Later, you'll retrieve the text for display simply by using the key associated with the text.

Note the string contains formatting information (for example, `/r`) to indicate returns and indentation. The final Moon Facts window will look like the following, once you've implemented the code to read and display the `Facts` string from the `Localizable.strings` file:



5. Save the Moon Travel project.

Create the About Window

In this section, you'll create an About window that displays configuration information read from the application's bundle. An About window is a standard element in a Macintosh application that, at a minimum, displays the application's icon, title, version, and copyright information. On Mac OS X, the About window opens when the user chooses About from the application menu.

A Mac OS X application stores configuration information in two places inside the bundle: an XML file called `Info.plist` and a localized string file called `InfoPlist.strings`. The section "Project Builder Items and Groups" (page 24) introduced `InfoPlist.strings` as a file in the Resources group in Project Builder. You should use the `InfoPlist.strings` file to store information that needs to be translated because it will be seen by users. The Get Info string, for example, is seen by users in the About window you'll create for the Moon Travel application.

A bundle can contain several `InfoPlist.strings` files, each stored in a different localization directory, such as `English.lproj` and `Japanese.lproj`, along with other localized resources. When you create an About window, you can read the version and other strings from the localization directory appropriate for the language used in the application. The Moon Travel project has one localization directory, `English.lproj`. Although it is possible to add other directories, you won't add any in this tutorial.

To create an About window, follow these steps:

1. "Create a New Window for the About Box" (page 51)
2. "Create Content for the About Window" (page 55)

Create a New Window for the About Box

1. Make Interface Builder active by clicking its icon in the Dock.
2. Create a window.

   This is the window you'll use to display the application's configuration information.

   Click the Windows button in the Carbon palette, then drag the icon of the window on the left to the Finder.
3. Name the window.

   In the Instances pane of the main.nib window, double-click the word "Window" that's under the icon of the window you just created, type `AboutWindow`, and press Return.
4. Name the title bar of the window `About Moon Travel`.

   With the window active, choose Inspector from the Tools menu, choose Attributes from the Window Inspector pop-up menu, type `About Moon Travel` in the Title text field, and press Return.
5. Set the Window class as Document.

   With the new window active, choose Inspector from the Tools menu.
6. Set the window's controls.

   In the Control group, make sure Close Box is the only option selected.

7. Set the window's attributes.

   In the Attribute group, make sure Standard Handler is the only option selected.

8. Resize the window to 264 by 165 pixels.

   This size allows for the application icon, application title, and two lines of configuration information. It also takes into account the spacing suggested in the *Aqua Human Interface Guidelines.*

   Choose Size in the Window Inspector pop-up menu, choose Width/Height from the right Content Rect pop-up menu, and type `264` for width and `165` for height.

9. Add the Moon Travel application icon to the Interface.

   You can use the same PICT you used for the main window. Follow the instructions in "Add Items to the Main Window" (page 31) for adding a PICT box to the interface. But in this instance, make sure you use the dimensions 64 by 64 when you size the PICT box. Then follow the instructions in "Add a Picture of the Moon to the Project" (page 41) for dragging a PICT resource to the PICT box.

10. Choose Show Layout Rectangles from the Layout menu.

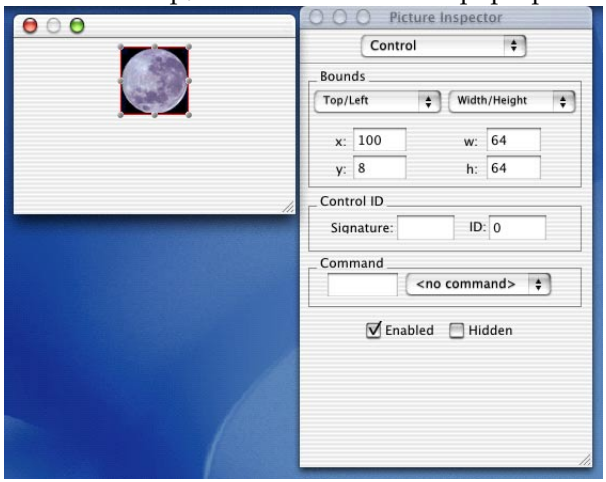
    You can align items more precisely using the edges of the layout rectangles.

11. Adjust the spacing of the Moon Travel application icon so it follows the recommendations in the *Aqua Human Interface Guidelines.*

    The current guidelines call for a spacing of 8 pixels from the top icon to the bottom of the title bar.

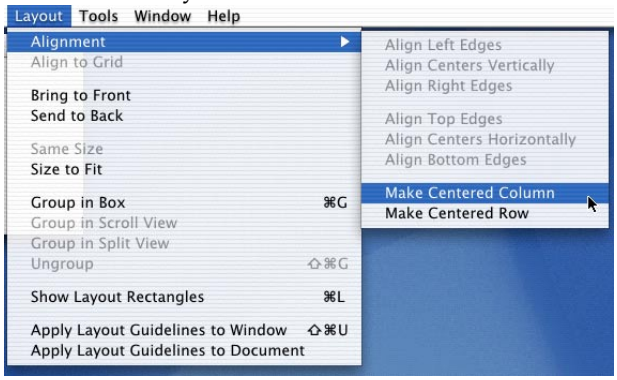
    With the icon selected, choose Inspector from the Tools menu, then choose Control from the pop-up menu.

Choose Top/Left from the Bounds pop-up menu and type `8` for the y-value.



12. Center the Moon Travel application icon.

    With the application icon selected, choose Alignment > Make Centered Column from the Layout menu.



13. Add a static text item below the application icon.

    This is the application title.

    From the Controls palette, drag the object named Static Text to the area below the icon.

14. Enter `Moon Travel` as the static text field's value.

    Double-click the field so a blinking insertion point appears. Type `Moon Travel` and press Return.

15. Adjust the spacing of the Moon Travel text so it follows the recommendations in the *Aqua Human Interface Guidelines.*

    The current guidelines call for the top of the Moon Travel text's layout rectangle to be 12 pixels from the bottom of the Moon Travel application icon.

    You actually need to calculate the spacing from the top of the window to the top of the Moon Travel text layout rectangle. That spacing is 8 + 64 + 12, or 84. Eight pixels for the space between the top of the window and the top of the application icon, 64 pixels for the height of the application icon, and 12 pixels for the space between the bottom of the application icon and the top of the Moon Travel text layout rectangle.

    With the Moon Travel static text field selected, choose Inspector from the Tools menu, then choose Control from the pop-up menu

    Choose Top/Left from the Bounds pop-up menu and type `84` for the y-value.

16. Center the Moon Travel text and adjust the spacing.

    With the Moon Travel static text field selected, choose Alignment > Make Centered Column from the Layout window.

17. Turn off the Alignment rectangles.

    Choose Hide Alignment Rectangles from the Layout menu.

    The About window should look like this:

18. Position the About window where you'd like it to appear when the user opens it.

    Where you position the window in Interface Builder determines its position when the user opens it.

Create Content for the About Window

You need to modify the `InfoPlist.strings` file in Project Builder so the version information is correct. Later you'll write code that reads the `InfoPlist.strings` file and writes the `CFBundleGetInfoString` to the About window.

1. Make the Moon Travel project window active.

   Click the Project Builder icon in the Dock.

2. Open the `InfoPlist.strings` file.

   Open Resources group and click `InfoPlist.strings`. You should see three strings that Project Builder added for you automatically.

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

   CFBundleName = "MoonTravel";
   CFBundleShortVersionString = "MoonTravel version 0.1";
   CFBundleGetInfoString = "MoonTravel version 0.1, Copyright 2000
   MyGreatSoftware.";
   ```

3. Modify the Get Info and Short Version strings so the version is 1.0.

4. Delete `MoonTravel` from the Get Info string.

5. Replace `MyGreatSoftware` with `Apple Computer, Inc.` or insert your company's name.

   The Get Info string should look like this after you edit it:

   ```
   "Version 1.0, Copyright 2000 Apple Computer, Inc.";
   ```

6. Save the Moon Travel project.

After you've implemented the code to read and display the `CFBundleGetInfoString`, the final About window will look like this:

Next you'll go back to Interface Builder to add and modify the main menu for the Moon Travel application.

Using Interface Builder to Add Menu Items

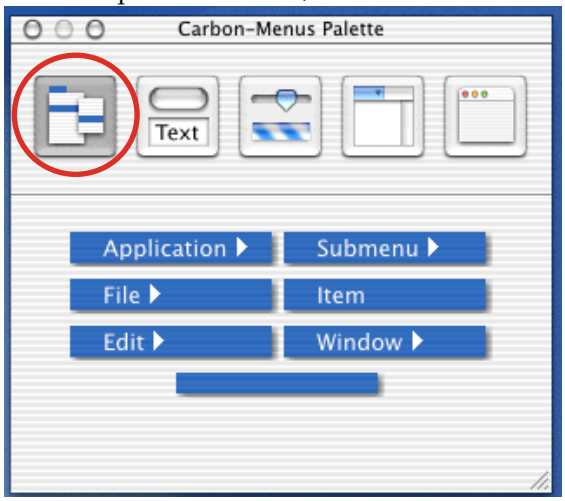
In this chapter, you'll add a menu with two items that issue the same commands as the Compute Travel Time and Moon Facts buttons, and you won't need to write any code to do it. You'll also add commands for two menu items that are created automatically for you: About Moon Travel and Moon Travel Help.

Add a Submenu to the Main Menu

In this section you'll use the Carbon-Menus Palette to add a Moon menu to the menu bar.

1. Click the Interface Builder icon in the Dock to make it active.

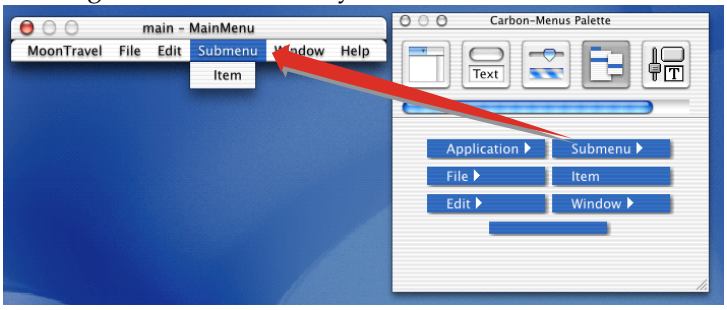
2. In the palette window, click the Menus button.



The Menus palette contains seven elements:

- The Application, File, Edit, and Window elements are fully-loaded menus that you can drop into your application. Note that these menus were added automatically when you created the Moon Travel application.
- The Submenu element can be either a top-level menu that you add to your menu bar or a hierarchical menu that you add to another menu.
- The Item element is a single menu item that you can add to any menu.
- The blank element is a separator that you can add to any menu.

3. Drag a Submenu item to your menu window, between Edit and Window.



4. Name the menu `Moon`.

Double-click the word Submenu, type `Moon`, and press Return.

Add a Compute Travel Time Menu Item

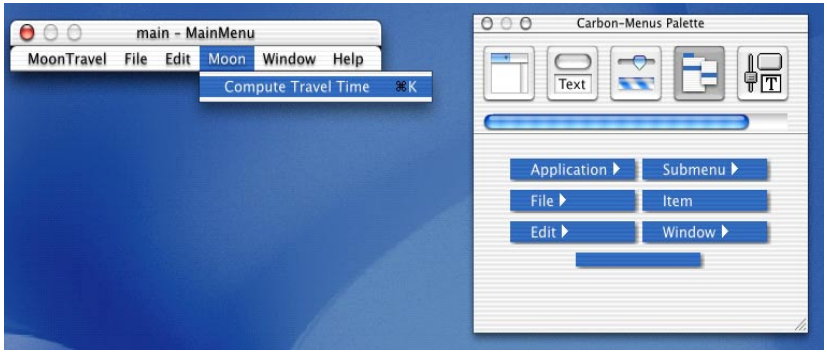
In this section, you'll add the Compute Travel Time command to the menu and give it a command-key equivalent.

1. If the Moon menu isn't open, click the word Moon.
2. Name the menu item `Compute Travel Time`.

   Double-click the menu item, type `Compute Travel Time`, and press Return.
3. Assign Command-K as the `Compute Travel Time` menu item's command-key equivalent.

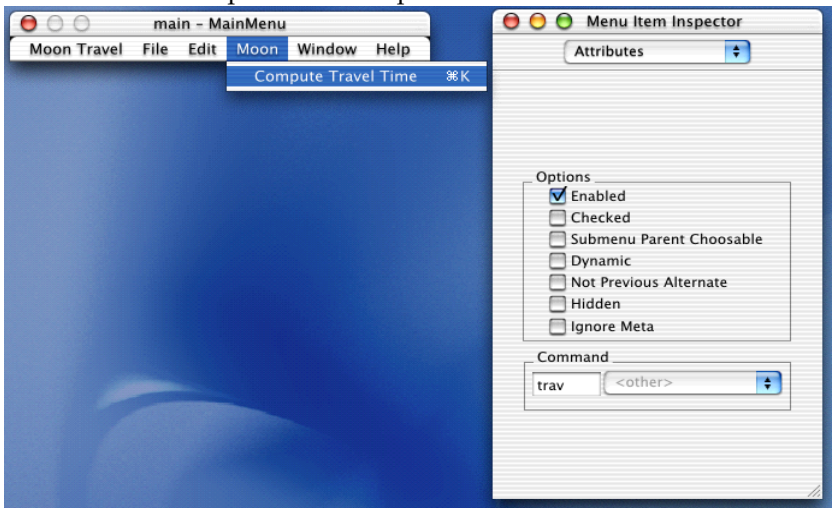
   Double-click to the right of `Compute Travel Time` so a box appears and type `Command-K`.

   You may have to try double-clicking a few places before you succeed. Try double-clicking from the right edge of the word `Time` all the way to the right edge of the menu.



4. Enter `trav` as the menu item's command.

Choose Inspector from the Tools menu, then choose Attributes from the pop-up menu. In the Command section, type `trav` in the text field. This is the same command you added when you created the Compute Travel Time button. The user gets the same result whether they choose Compute Travel Time from the Moon menu or press the Compute Travel Time button in the main window.

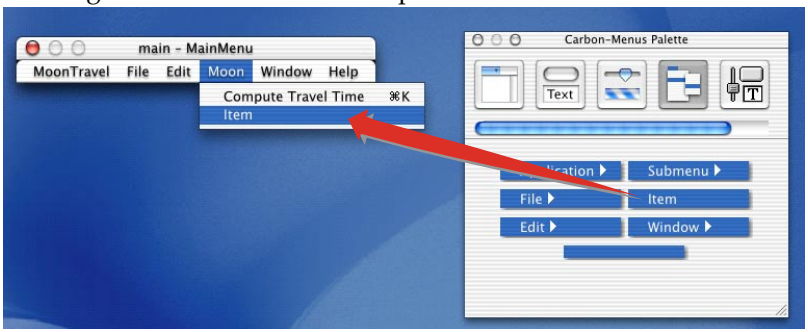


When the user chooses Compute Travel Time, the Carbon Event Manager calls your handler for the `trav` command. You'll define the handler in the chapter "Writing the Event Handlers and Other Code" (page 65).

Add a Moon Facts Menu Item

In this section, you'll add the Moon Facts command to the Moon menu. After you drag a new item to the Moon menu, the procedure for naming it, assigning a command-key equivalent, and associating a command with it is similar to what you did to create the `Compute Travel Time` command.

1. Drag an item from the Menus palette to the Moon menu.



2. Name the menu item `Moon Facts`.

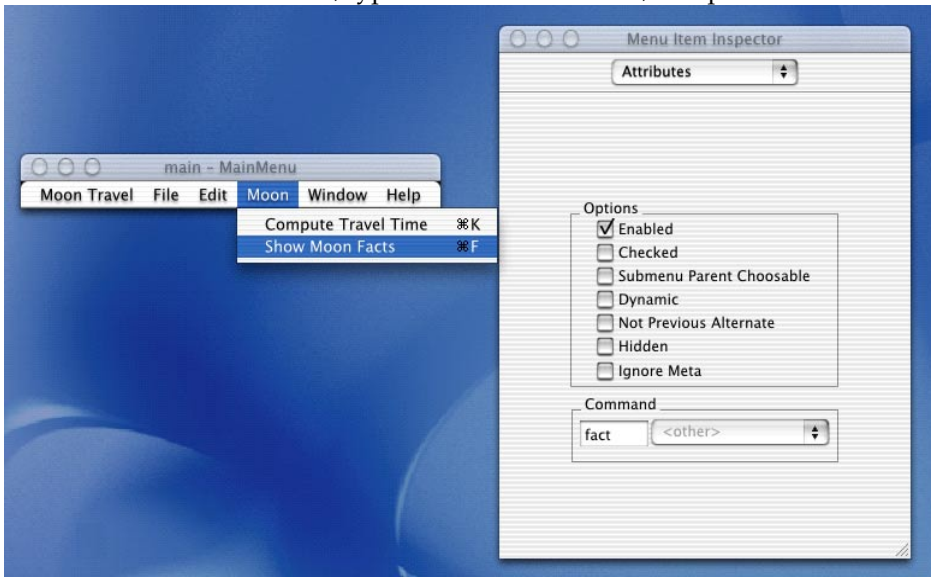
   Double-click the menu item so a blinking insertion point appears, type `Moon Facts`, and press Return.

3. Assign `Command-F` as the `Moon Facts` menu item's command-key equivalent.

   Double-click to the right of the Moon Facts menu item and type `Command-F`.

4. Enter `fact` as the menu item's command.

   Choose Attributes from the pop-up menu at the top of the Menu Item Inspector. In the Command section, type `fact` in the text field, and press Return.

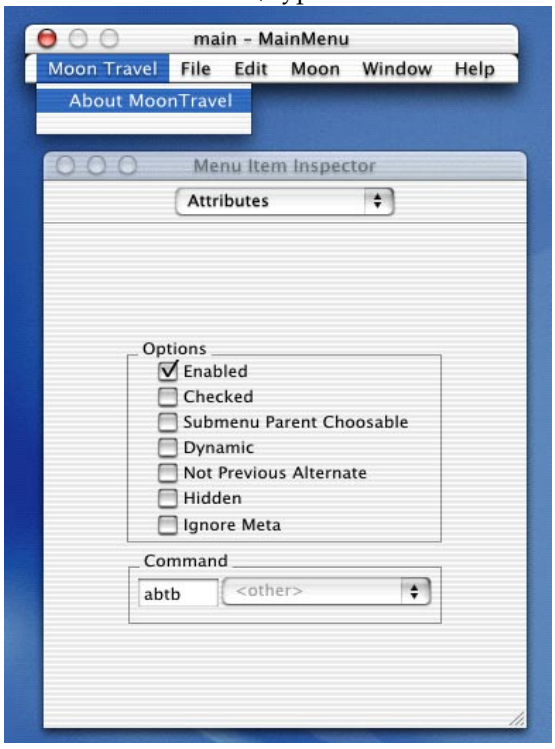


Add an About Command

The About menu item is automatically included in the main menu. All you need to do is add a command that opens the About window you created in the section "Create the About Window" (page 50).

- Enter `abtb` as the `About MoonTravel` menu item's command.

  From the pop-up menu at the top of the Inspector, choose Attributes. In the Command section, type `abtb` in the text field, and press Return.

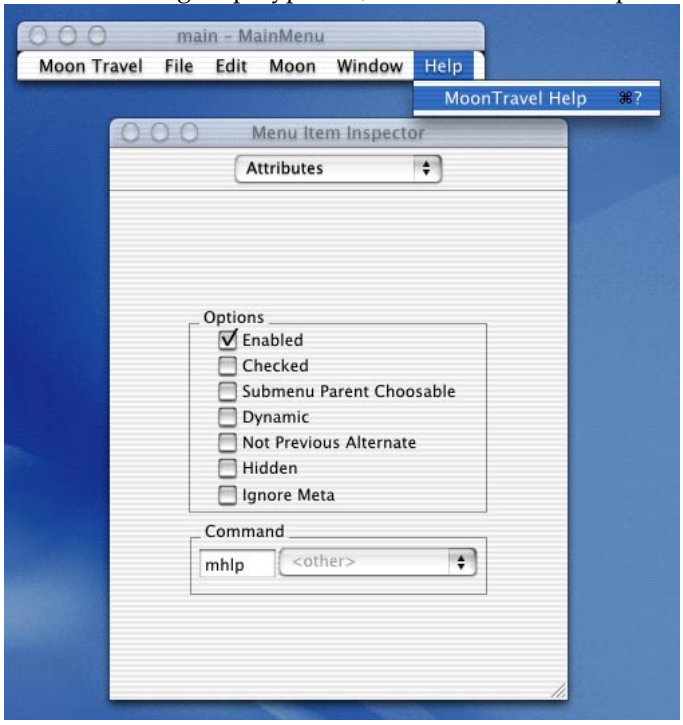


Add a Help Menu Command

The Help menu item is included automatically in the main menu. You need to add a command to invoke a function in your code that will call the Apple Help Manager function to open Moon Travel Help. (You'll write the function in "Adding the Moon Travel Help Book" (page 85).

1. Click the Help menu item.
2. Click MoonTravel Help.
3. Enter `mhlp` as the menu item's command.

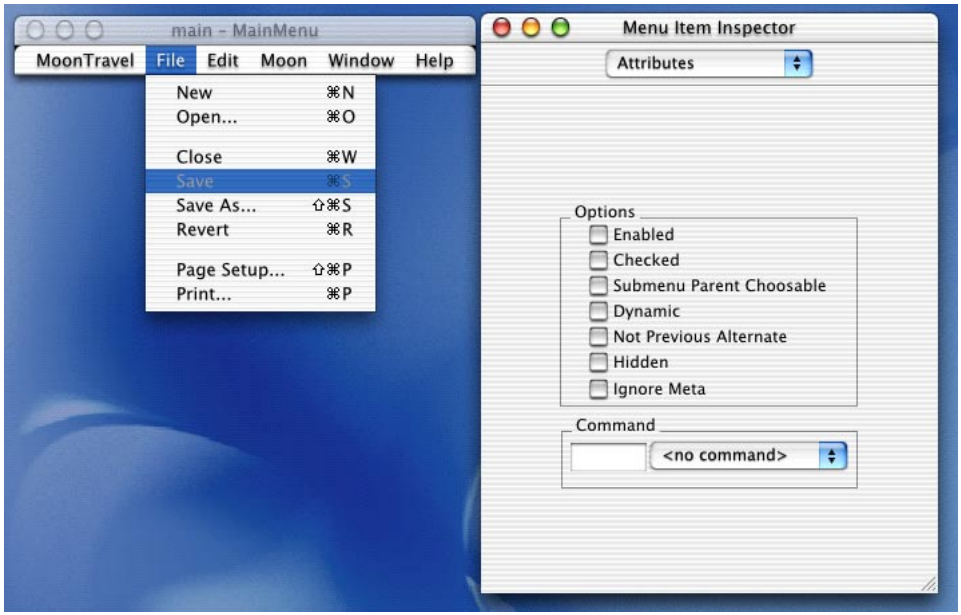
   Choose Attributes from the pop-up menu in the Menu Item Inspector. In the Command group, type `mhlp` in the text field and press Return.



Disabling Menu Items

A number of menu items (such as Save and Save As) are automatically included in the main menu, but are not needed in the Moon Travel application. Disable unused items by following the steps below:

1. Click on the item you want to disable.
2. Open the Inspector for the item.
3. Make sure Enabled is not selected.



4. Quit Interface Builder.

   You're done with the interface. Next, you'll write the code for the Moon Travel application.

CHAPTER 7

Writing the Event Handlers and Other Code

In this chapter, you'll see what an application that uses the Carbon Event Manager looks like. Then you'll write the code that handles events for each window you created, including functions that handle the `trav`, `fact`, `abtb`, and `mhlp` commands. For readability, most error-checking code has been omitted from the code listings.

1. "Look at the Existing Code" (page 65)
2. "Install the Window Event Handlers" (page 68)
3. "Declare Constants and Global Variables" (page 66)
4. "Write the Main Window Event Handler" (page 69)
5. "Write the Moon Facts Window Event Handler" (page 78)
6. "Write the About Window Event Handler" (page 79)
7. "Declare the Window Event Handlers" (page 80)
8. "Add and Modify Code to Create the Interface" (page 81)
9. "Build, Run, and Test the Application" (page 82)

Look at the Existing Code

Before you start adding code to the `main.c` file that Project Builder created for you, take a look at the code already in the file.

1. Make the Moon Travel project active by clicking the Project Builder icon in the Dock.

2. Click `main.c` in the project window's file list.

   If you can't see the `main.c` file, click the Files tab and open the Sources group.

   You need only six functions to write an application that uses the Carbon Event Manager. Veteran Mac programmes will notice that they don't need to initialize any Toolbox managers, nor do they need to write an event loop; all that's handled for them automatically.

   The functions listed below are from the `main.c` file in your project, the error checking and comments have been removed.

```
CreateNibReference (CFSTR (“main”), &nibRef);
SetMenuBarFromNib (nibRef, CFSTR("MainMenu"));
CreateWindowFromNib (nibRef, CFSTR (“MainWindow”), &window);
DisposeNibReference (nibRef);
ShowWindow (window);
RunApplicationEventLoop ();
```

   Here's what the statements do:

   - `CreateNibReference` searches your application's package for a file called `main.nib` and opens it.
   - `SetMenuBarFromNib` and `CreateWindowFromNib` set up the menu bar and main window from the nib file.
   - `DisposeNibReference` closes the nib file.
   - `ShowWindow` displays the main window, since it was set up to be hidden by default.
   - `RunApplicationEventLoop` runs the main event loop.

Declare Constants and Global Variables

In this section you'll declare the constants and global variables used in the Moon Travel application. Many of these constants represent the commands, IDs, and signatures you assigned to interface objects when you created the interface.

Copy the following declarations to the `main.c` file, at the top, just after the statement `#include <Carbon/Carbon.h>`:

```
// Define constants for the commands, IDs, and signatures used
// in the interface. Make sure the values match those you assigned when
// you set up the interface in Interface Builder.
#define kComputeCommand      'trav'
#define kComputeSignature    'trav'
#define kShowMoonFactsCommand    'fact'
#define kOpenAboutBoxCommand     'abtb'
#define kOpenHelpCommand         'mhlp'
#define kTravelTimeFieldID                   129
#define kModeOfTransportationButtonGroupID   130

// Define constants to use in the moon travel time computation.
#define kHoursPerDay     24
#define kDistanceToMoon 384467  // kilometers

// Define constants used for About Box layout
#define kIconHeight          64      // height of Moon Travel app icon
#define kAboveIconSpacing   8        // must be 8 pixels above the app icon
#define kBelowIconSpacing   12       // must be 12 pixels below the app icon
#define kAppTitleHeight     17      // height of rect. enclosing app title
#define kBetweenLineSpacing 8        // spacing between lines of text

// Define constants to identify the mode of transportation
#define kFootMode                1
#define kCarMode                 2
#define kCommercialJetMode       3
#define kApolloSpacecraftMode    4

// Global window references.
WindowRef       gMainWindow,
                gMoonFactsWindow,
                gAboutBoxWindow;
```

Install the Window Event Handlers

In this section, you'll declare event type specifiers associated with each window event handler, then install the event handlers for the Main, Moon Facts, and About windows. An event type is defined as an event class paired with an event kind. For more information see "Event Types" (page 12).

1. Declare the event type specifiers for the main window.

   When you install an event handler, the event type specifier tells the Carbon Event Manager when to call the handler. The main window event handler takes care of command events.

   Copy the following to the `main` function, after the declaration `OSStatus err`:

```
EventTypeSpec       mainSpec = { kEventClassCommand,
                                 kEventProcessCommand};
```

Note: Although each window event handler declared in the Moon Travel application takes care of one type of event, event handlers can handle multiple events. For handlers that take care of more than one event, you can declare an array of event specifiers.

2. Declare the event type specifiers for the Moon Facts window.

   The Moon Facts window handler takes care of an event in the close button.

   Copy the following to the `main` function, after the declaration for the main window event specifier:

```
EventTypeSpec    moonSpec = { kEventClassWindow,
                              kEventWindowClose };
```

3. Declare the event type specifiers for the About window.

   The About window handler takes care of an event in the close button.

   Copy the following to the `main` function, after the declaration for the Moon Facts window event specifier:

```
EventTypeSpec aboutSpec = { kEventClassWindow,
```

```
                                    kEventWindowClickContentRgn };
```

4. Install the event handlers.

   After the line `DisposeNibReference(nibRef)`, add the following code:

```
InstallWindowEventHandler (gMainWindow,
            NewEventHandlerUPP (MainWindowEventHandler),
            1, &commandSpec, (void *) gMainWindow, NULL);
InstallWindowEventHandler (gMoonFactsWindow,
            NewEventHandlerUPP (MoonFactsWindowEventHandler),
            1, &moonSpec, (void *) gMoonFactsWindow, NULL);
InstallWindowEventHandler (gAboutBoxWindow,
            NewEventHandlerUPP (AboutBoxEventHandler),
            1, &aboutSpec, (void *) gAboutWindow, NULL);
```

   `InstallWindowEventHandler` tells the Carbon Event Manager to call the window's event handler whenever a specified event type happens in the window. The parameters to the function `InstallWindowEventHandler` are the following:

   - the target of the event handler—in this case, the window to which the handler should be registered
   - a pointer to the window event handler—in this case, the function `NewEventHandlerUPP` returns a pointer to the specified function
   - the number of events for which the handler is registered
   - a pointer to the event types handled by the event handler
   - a value that's passed on to the window event handler when the Carbon Event Manager calls the handler—in this case, a pointer to the window to which the handler is registered
   - a pointer to an event handler reference—in this case, `NULL`

Write the Main Window Event Handler

In this section, you'll write the event handler for the main window and the three functions called by the main window event handler:

- `ComputeTravelTimeCommandHandler`
- `MoonFactsCommandHandler`
- `AboutBoxCommandHandler`

When you set up the window attributes for the main window in the section "Set Attributes for the Main Window" (page 42), you selected Standard Handler as one of the window's attributes. This means that the Carbon Event Manager handles all of the usual window behavior for you—responding to drag events, minimizing the window, quitting the application, and so forth. When an event associated with the main window occurs, the Carbon Event Manager passes the event to your application's main window event handler. If the event handler doesn't handle the event, it can pass back the result code `eventNotHandledErr`, and the Carbon Event Manager handles the event if it can. Otherwise, the event is ignored.

The handler you'll write in this section needs to take care of events that aren't standard window events, namely, responding to button presses for the Compute Travel Time and Moon Facts buttons.

The application-specific commands in the main menu are treated by the Carbon Event Manager as events associated with the main window. So in addition to handling Compute Travel Time and Moon Facts button presses, the Main window event handler must process the main menu commands you created for the About and Help menu items. From the Carbon Event Manager's perspective, it doesn't matter whether the user issues the Compute Travel Time and Moon Facts commands by choosing a menu command or by clicking a button in the main window. The main window event handler in Listing 7-1) takes care of the command regardless of its origin.

Copy the function in Listing 7-1 into the main program (`main.c`), after the `main` function.

Listing 7-1 The main window event handler

```
pascal OSStatus MainWindowEventHandler (EventHandlerCallRef handlerRef,
                        EventRef event, void *userData)
{
    OSStatus        result   = eventNotHandledErr;                    // 1
    HICommand       command;
```

```
    GetEventParameter (event, kEventParamDirectObject, typeHICommand, NULL,
                                    sizeof (HICommand), NULL, &command); // 2
    switch (command.commandID)
      {
        case kComputeCommand:
              ComputeCommandHandler ((WindowRef) userData);            // 3
              result = noErr;
              break;
        case kShowMoonFactsCommand:
               MoonFactsCommandHandler (gMoonFactsWindow);             // 4
               result = noErr;
              break;
        case kOpenHelpCommand:
               AHGotoPage (CFSTR ("Moon Travel Help"), NULL, NULL);   // 5
               result = noErr;
             break;
        case kOpenAboutBoxCommand:
               AboutBoxCommandHandler (gAboutWindow);                 // 6
               result = noErr;
             break;
      }
    return result;
}
```

Here's what the function does:

1. The handler sets the value of `result` to `eventNotHandledErr`. This assures that if the event passed to the handler is not handled, the Carbon Event Manager handles the event if it can.
2. The Carbon Event Manager function `GetEventParameter` gets the command ID associated with the command. The command IDs are the four-character codes you assigned to the buttons and commands in the interface and declared as constants in the section "Declare Constants and Global Variables" (page 66).
3. `Compute Command Handler` computes travel time and displays the result. You must write this function.
4. `MoonFactsCommandHandler` shows the Moon Facts window and displays information about the moon . You must write this function.

5. The Apple Help Manager function `AHGoToPage` opens the Help Viewer application to the Moon Travel Help table of contents. The first parameter must be a string that matches the title of the help book's table of contents page. This is the title that is set by the HTML `Title` tag.The function `AHGoToPage` assumes you have a properly constructed help book that is registered with the operating system. See "Adding the Moon Travel Help Book" (page 85) for instructions on adding and registering the Moon Travel Help book provided with this tutorial.

   If the Help book or page aren't found by the Apple Help manager, the function `AHGoToPage` opens the Help Viewer to a blank page.

6. `AboutBoxCommandHandler` shows the About window and displays version information for the Moon Travel application. You must write this function.

Write the Compute Travel Time Command Handler

In this section, you'll write a function that computes travel time. The function

1. reads the value of the radio button group
2. chooses a calculation based on the radio button group, then makes the calculation
3. converts the numerical result to a string
4. writes the string to the travel time field

Copy the function in Listing 7-2 into the main program, after the function `MainWindowEventHandler`.

Listing 7-2 A handler that computes travel time based on the mode of transportation selected by the user

```
pascal void ComputeCommandHandler (WindowRef window)
{
  ControlHandle      modeOfTransportButtonGroup,
                     travelTimeField;
  ControlID   modeOfTransportControlID = { kComputeSignature,
                         kModeOfTransportationButtonGroupID };
  ControlID  travelTimeControlID = { kComputeSignature,
                         kTravelTimeFieldID };
  CFStringRef    text;
```

```
  double         travelTime;
  SInt32         transportModeValue;

  GetControlByID (window, &modeOfTransportControlID,                    // 1
                          &modeOfTransportButtonGroup );
  GetControlByID (window, &travelTimeControlID, &travelTimeField );
  transportModeValue = GetControl32BitValue (modeOfTransportButtonGroup);
                                                                        // 2
 switch (transportModeValue)                                            // 3
   {
     case kFootMode:
        // Foot - good walking time is 4 miles per hour
        travelTime = (kDistanceToMoon/(4.0/0.62))/kHoursPerDay;
        break;
     case kCarMode:
        // Car - 70 miles per hour on the highway, no speed limit in space!
        travelTime = (kDistanceToMoon/(70/0.62))/kHoursPerDay;
        break;
     case kCommercialJetMode:
        // Commercial Jet - 600 mile per hour.
        travelTime = (kDistanceToMoon/(600.0/0.62))/kHoursPerDay;
        break;
     case kApolloSpacecraftMode:
        // Apollo 11 took 4 days to get to the moon.
        // Use that as the basis for comparison with
        travelTime = 4;
        break;
     default:
         travelTime = 0;
        break;
   }
  text = CFStringCreateWithFormat( NULL, NULL, CFSTR("%g"), travelTime);// 4
  SetControlData( travelTimeField, 0, kControlEditTextCFStringTag,    // 5
                            sizeof (CFStringRef), &text);
  CFRelease (text);                                                     // 6
  DrawOneControl (travelTimeField);                                     // 7
}
```

Here's what the function does:

1. The Control Manager function `GetControlByID` gets the control handle associated with the Mode of Transportation radio button group and the Travel Time field.
2. The Control Manager function `GetControl32BitValue` returns a value that indicates which radio button in the radio button group is selected.
3. Calculations are based on the mode of transportation selected, but do not take into account the effects of gravity and inertia. Miles are converted to kilometers by dividing by 0.62.
4. The Core Foundation String Services function `CFStringCreateWithFormat` converts the floating point number to a `CFString`. The "printf-style" format string `%g` indicates how the string should be formatted.
5. The Control Manager function `SetControlData` sets the Travel Time text field to the value of the string, but does not draw the field.
6. The Core Foundation Base Services function `CFRelease` releases the memory associated with the `CFString`.
7. The Control Manager function `DrawOneControl` redraws the Travel Time text field with the new string.

Write the Moon Facts Command Handler

In this section, you'll write a function that shows the Moon Facts window. The function

- shows the Moon Facts window if it is not visible
- expands the window if the window is already visible but in the Dock
- writes a string from the `Localizable.strings` file to the window.

Copy the function in Listing 7-3into the main program, after the function `ComputeCommandHandler`.

Listing 7-3 A function that shows the Moon Facts window

```
pascal void MoonFactsCommandHandler (WindowRef window)
{
   GrafPtr          originalPort;
```

```
    Rect              bounds;
    CFStringRef       text;

     GetPort (&originalPort);                                          // 1
    if (IsWindowCollapsed (window))                                    // 2
         CollapseWindow (window, FALSE);
     {
     else
         ShowHide (window, TRUE);
         SelectWindow (window);                                        // 3
      }
     SetPortWindowPort (window);                                       // 4
     GetWindowPortBounds (window, &bounds);                            // 5
     EraseRect (&bounds);
     text = CFCopyLocalizedString (CFSTR("Facts"),CFSTR("string"));    // 6
     SetRect (&bounds, bounds.left, bounds.top, bounds.right, bounds.bottom);
                                                                       // 7
     TXNDrawCFStringTextBox (text, &bounds, NULL, NULL);               // 8
     SetPort (originalPort);                                           // 9
}
```

Here's what the function does:

1. The QuickDraw Manager function `GetPort` saves the current graphics port. You need to restore this later.
2. The Window Manager function `IsWindowCollapsed` checks to see if the window is in the Dock. If it is, the `CollapseWindow` function expands the window to its normal size. Otherwise the function `ShowWindow` makes the window visible.
3. The Window Manager function `SelectWindow` makes the window active and frontmost.
4. The Window Manager function `SetPortWindowPort` sets the graphics port to the active window.
5. The Window Manager function `GetWindowPortBounds` gets the bounds of the active window while the QuickDraw function `EraseRect` makes sure the window is empty.
6. The Core Foundation Bundle Services function `CFCopyLocalizedString` returns the string associated with the key "Facts" from the `Localizable.strings` file. You added this string to the `Localizable.strings` file in the section "Create Content for the Moon Facts Window" (page 49).

7. The QuickDraw Manager function `SetRect` sets the bounds of the rectangle into which you draw the string so the rectangle is the same area as the Moon Facts window.
8. The Multilingual Text Engine function `TXNDrawCFStringBox` draws static text into the rectangle defined by the function `SetRect`. The last two parameters to `TXNDrawCFStringBox` define style and display options. Pass `NULL` to use the options associated with the current graphics port.
9. The function `SetPort` restores the original graphics port.

Write the About Window Command Handler

In this section, you'll write a function that shows the About window. The function

- shows the About Box window
- writes a string from the `Info.plist` file to the window

Copy the function in Listing 7-4 into the main program, after the function `MoonFactsCommandHandler`.

Listing 7-4 A function that shows the About window

```
pascal void AboutBoxCommandHandler (WindowRef window)
{
    GrafPtr                     originalPort;
    Rect                        bounds;
    CFStringRef                 text;
    CFBundleRef                 appBundle;
    SInt16                      spacingAdjustment;
    TXNTextBoxOptionsData        displayOptions;

    GetPort (&originalPort);                                        // 1
    ShowHide (window, TRUE);                                        // 2
    SelectWindow (window);
    SetPortWindowPort (window);                                     // 3
    GetWindowPortBounds (window, &bounds);                          // 4
    appBundle = CFBundleGetMainBundle ();                           // 5
    text = (CFStringRef) CFBundleGetValueForInfoDictionaryKey (appBundle,
                                    CFSTR("CFBundleGetInfoString"));  // 6
```

```
    if ((text == CFSTR(" ") ) || (text== NULL))
           text = CFSTR("Nameless Application.");                    // 7
    spacingAdjustment = kAboveIconSpacing + kIconHeight + kBelowIconSpacing
                           + kBetweenLineSpacing;                    // 8
    SetRect (&bounds, bounds.left, bounds.top + spacingAdjustment,
                                bounds.right, bounds.bottom);         // 9
    displayOptions.optionTags = kTXNSetFlushnessMask;                //10
    displayOptions.flushness = kATSUCenterAlignment;
   TXNDrawCFStringTextBox (text, &bounds, NULL, &displayOptions);    //11
    SetPort (originalPort);                                          //12
}
```

Here's what the function does:

1. The QuickDraw Manager function `GetPort` saves the current graphics port. You need to restore this later.
2. The Window Manager function `ShowWindow` makes the About window visible. The function `SelectWindow` makes the window active and frontmost.
3. The Window Manager function `SetPortWindowPort` sets the graphics port to the active window.
4. The Window Manager function `GetWindowPortBounds` gets the bounds of the active window.
5. The Core Foundation Bundle Services function `CFBundleGetMainBundle` returns the bundle associated with the Moon Travel application. You need to pass the bundle as a parameter to the function in the next statement.
6. The Core Foundation Bundle Services function `CFBundleGetValueForInfoDictionaryKey` returns the string associated with the key `CFBundleGetInfoString`. This is the string you modified in the section "Create Content for the About Window" (page 55). The function `CFBundleGetValueForInfoDictionaryKey` takes a `CFString` object as a parameter, so you must use the function `CFSTR` to convert the Get Info string to a `CFString` object.
7. To catch possible errors, if the string `CFBundleGetInfoString` doesn't exist, then set the value of `text` to "Nameless Application."
8. You'll declare the constants in this statement later, in the section "Declare Constants and Global Variables" (page 66). You need the spacing adjustment to adjust the area of the rectangle into which you'll draw the Get Info string. The

spacing adjustment takes into account the size of PICT of the Moon and the application title as well as the spacing recommended by the *Aqua Human Interface Guidelines*.

9. The QuickDraw Manager function `SetRect` sets the bounds of the rectangle into which you draw the string. The top of the rectangle is adjusted to assure the text is drawn below the application icon and title you added in the section "Create a New Window for the About Box" (page 51).
10. Set `displayOptions` to pass to the Multilingual Text Engine (MLTE) function `TXNDrawCFStringBox`. The option tag `kTXNSetFlushnessMask` is an MLTE constant that indicates to set the flushness bit. The flushness value `kATSUCenterAlignment` is an Apple Type Services for Unicode Imaging (ATSUI) constant that indicates the text should be drawn centered.
11. The function `TXNDrawCFStringBox` draws static text into the rectangle defined by the function `SetRect`. The third parameter is set to `NULL` to indicate the style associated with the current graphics port.
12. The function `SetPort` restores the original graphics port.

Write the Moon Facts Window Event Handler

In this section, you'll write the event handler for the Moon Facts window. When you set up the window attributes for the Moon Facts window in the section "Create the Moon Facts Window" (page 47), you selected Standard Handler as one of the window's attributes. You can let the Carbon Event Manager handle every event associated with this window—dragging, activating, minimizing, expanding the window from the Dock—except closing. The standard close event would dispose of the window. But because the Moon Travel application creates the Moon Facts window once (when the application opens) and disposes of the window once (when the application quits), the standard close event isn't appropriate. The Moon Travel application must open and close the Moon Facts window by showing and hiding the window.

Copy the function in Listing 7-5 into the main program, after the function `MainWindowEventHandler`.

Listing 7-5 The Moon Facts window event handler

```
pascal OSStatus MoonFactsWindowEventHandler (
                        EventHandlerCallRef myHandler,
                       EventRef event, void *userData)
{
    OSStatus         result = eventNotHandledErr;
    UInt32           eventKind;

    eventKind = GetEventKind (event);                                   // 1
    if (eventKind == kEventWindowClose)                                 // 2
     {
            ShowHide ( (WindowRef) userData, FALSE);                    // 3
            SelectWindow (gMainWindow);                                 // 4
            result =  noErr;
      }
   return result;
}
```

Here's what the function does:

1. The Carbon Event Manager function `GetEventKind` returns the kind of the event.
2. If the event is a window close event, then handle it, otherwise return the result code `eventNotHandledErr` to indicate the Carbon Event Manger should handle the event.
3. The Window Manager function `ShowHide` hides the Moon Facts window.
4. The Window Manager function `SelectWindow` makes the main window active and frontmost. You declared the global variable `gMainWindow` in the section "Declare Constants and Global Variables" (page 66).

Write the About Window Event Handler

In this section, you'll write the event handler for the About window. This handler is similar to the Moon Facts window event handler in that the only event the handler takes care of is closing the About window.

Copy the function in Listing 7-6 into the main program, after the function `MoonFactsWindowEventHandler`.

Listing 7-6 The About window event handler

```
pascal OSStatus AboutBoxWindowEventHandler (EventHandlerCallRef handlerRef,
EventRef event, void *userData)
{
    OSStatus         result   = eventNotHandledErr;
    UInt32           eventKind;

    eventKind = GetEventKind (event);                                    // 1
    if ( eventKind == kEventWindowClickContentRgn)                       // 2
    {
            ShowHide ( (WindowRef) userData, FALSE);                     // 3
             result = noErr;
    }
    return result;
}
```

Here's what the function does:

1. The Carbon Event Manager function `GetEventKind` returns the kind of the event.
2. If the event is a click event in the content region of the window, then handle it, otherwise return the result code `eventNotHandledErr` to indicate the Carbon Event Manger should handle the event.
3. The Window Manager function `ShowHide` hides the About window.

Declare the Window Event Handlers

In this section, you'll create a function declaration for each of the window event and command handlers you've added to the `main.c` file.

Copy the declarations shown in Listing 7-7) to the `main.c` file, just after the constants and global variable declarations.

Listing 7-7 Function declarations for the window and command event handlers

```
// Function declarations for the window event handlers
pascal OSStatus MainWindowEventHandler(EventHandlerCallRef handlerRef,
                                       EventRef event, void *userData);
pascal OSStatus MoonFactsWindowEventHandler (EventHandlerCallRef myHandler,
                                       EventRef event, void *userData);
pascal OSStatus AboutBoxWindowEventHandler (EventHandlerCallRef handlerRef,
                                       EventRef event, void *userData);

// Function declarations for the command event handlers
pascal void ComputeCommandHandler ();
pascal void MoonFactsCommandHandler (WindowRef window);
pascal void AboutBoxCommandHandler (WindowRef window);
```

Add and Modify Code to Create the Interface

In this section you'll modify some of the code in the `main` function that's automatically provided by Project Builder and add additional code that constructs the interface from the `main.nib` file you modified in Interface Builder. You'll also add code to dispose of the windows when the application quits.

The code you'll be modifying and adding uses the the global window variables you declared in the section "Declare Constants and Global Variables" (page 66).

1. Modify the existing code that creates the main window.

   Replace the `CreateWindowFromNib` statement with the following:

   ```
   err = CreateWindowFromNib (nibRef, CFSTR ("MainWindow"), &gMainWindow);
   ```

   This edits the `CreateWindowFromNib` statement so it uses `gMainWindow` instead of `window`. You'll declare the global window variable `gMainWindow` later.

2. Add code to create the Moon Facts and About windows from the Interface Builder `main.nib` file.

   Copy the following statements to the `main` function in the `main.c` file, after the statement that creates the main window.

```
err = CreateWindowFromNib (nibRef, CFSTR("MoonFacts"), &gMoonFactsWindow);
err = CreateWindowFromNib (nibRef, CFSTR("AboutBox"), &gAboutWindow);
```

3. Modify the existing code to show the main window.

   The operating system creates windows in a hidden state. When the application launches, the main window should be visible, and the other windows hidden. To make the main window visible you need to edit the `ShowWindow` statement so it uses `gMainWindow` instead of `window`.

   Replace the `ShowWindow` statement with the following:

   ```
   ShowWindow (gMainWindow);
   ```

4. Make sure the windows are disposed of when the application quits.

   Add the following lines to the `main` function, just after the `RunApplicationEventLoop` statement:

   ```
   DisposeWindow (gMainWindow);
   DisposeWindow (gMoonFactsWindow);
   DisposeWindow (gAboutBoxWindow);
   ```

Build, Run, and Test the Application

The Moon Travel application is almost complete. You only need to add the Moon Travel Help book and the code necessary to register the help book and its title page. Before you proceed to "Adding the Moon Travel Help Book" (page 85) take a moment to build, run, and test the application.

1. Click the Build button in the upper-left corner of the Moon Travel project window.

   If the project does not build, check the messages. You may have introduced a typographical error. If you need information on using the built-in debugger, see *DebugApp: Debugging an Application with Project Builder*, available through the Mac OS X Developer Documentation website (see Developer Tools):

   http://developer.apple.com/techpubs/macosx/

2. Click the Run button in the upper-left corner of the project window.

3. Test the application.

   Press each button in the interface, then select each menu command you added. Does the interface look as you expect? If not, you may need to open Interface Builder and make changes. Do the buttons and menu commands operate as you expect? If not, you may need to check the code, the `Info.plist` and `Localizable.strings` files.

Note: If you choose Moon Travel Help from the Help menu, the Help Viewer should open to a blank page.

Adding the Moon Travel Help Book

This section describes how to add the Help book provided with the Moon Travel tutorial. (If you want to create your own Help book, follow the instructions in the Apple Help SDK, available from the Apple Developer's website.)

To add the Moon Travel Help book, you must do the following:

1. "Add the Moon Travel Help Book" (page 85)
2. "Modify the Application's Property List" (page 86)
3. "Create a Function to Register the Help Book" (page 87).
4. "Build, Run, and Test the Application" (page 88)

Add the Moon Travel Help Book

The Moon Travel Help book needs to be added to the localized folder, `English.lproj`, and then added as a resource to the Moon Travel application.

1. From the Finder, copy the Moon Travel Help folder to the `English.lproj` folder in the MoonTravel folder.
2. Open the Moon Travel project.
3. Add the Moon Travel Help folder to the Moon Travel project.

   Choose Add Files from the Project menu, select the Moon Travel Help folder, and click Open.

   Deselect "Recursively create groups for added folders," and click Add.

Project Builder adds the folder as a resource as long as you've deselected this option.

4. If you need to, drag the Moon Travel Help book folder to the Resources group.

Modify the Application's Property List

You need to place two entries in the application's property list so that the Apple Help Manager knows where to find the Moon Travel Help book. The first entry consists of a `CFBundleHelpBookFolder` key with a string value identifying the folder for your help book. The second entry consists of a `CFBundleHelpBookName` key with a string value specifying the help book's title, as defined in your title page file.

The property list must also contain a valid entry for the `CFBundleIdentifier`. The bundle identifier is used by Core Foundation Bundle Services to find the bundle that contains the application's resource.

1. Click the Targets tab, then click Moon Travel in the Targets list.
2. Click Application settings, then click Expert.
3. Create an entry for the Help book folder name.

   Click new sibling, type `CFBundleHelpBookFolder` as the property name and press Return. Then type `Moon Travel Help` as the property value and press Return.

4. Create an entry for the Help book name.

   Click New Sibling, type `CFBundleHelpBookName` as the property name, and press Return. Then type `Moon Travel Help` as the property value and press Return.

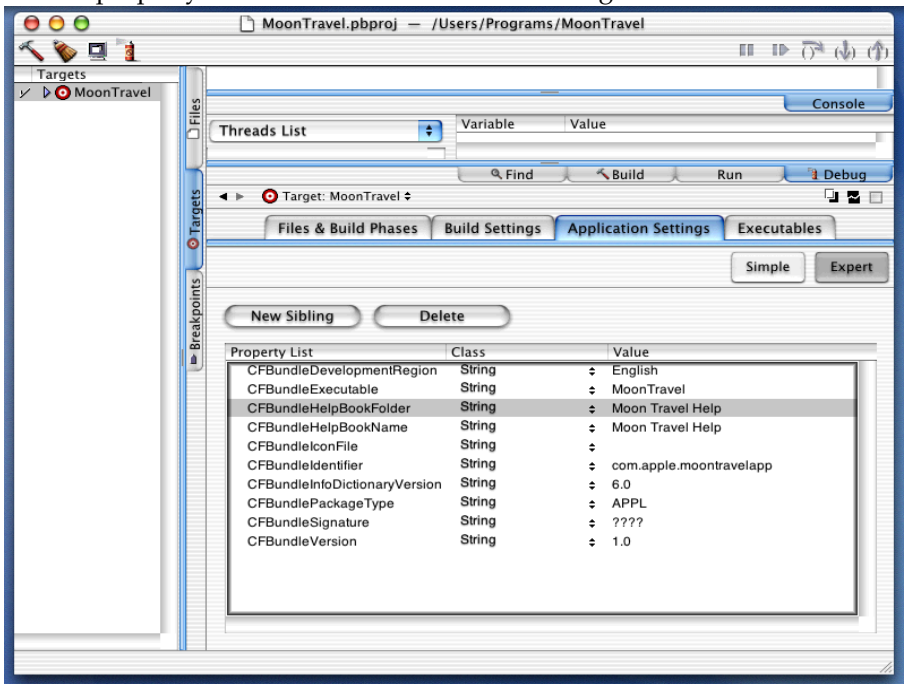
5. Create an entry for the bundle identifier name.

   To ensure that it's unique, the bundle identifier should be a Java-style package name; for example `com.mycompany.myapp` or `edu.ABCSchool.myapp`.

   Click New Sibling, type `CFBundleIdentifier` as the property name, and press Return.

   Then type `com.apple.moontravelapp` as the property value and press Return.

The property list should similar to the following:



Create a Function to Register the Help Book

Registering a help book causes it to appear in the Help Center.

1. Click the Files tab, then in the Sources group, click `main.c`.
2. Declare the function `RegisterMoonTravelHelp`.

   Add the following statement before the `main` function, just after the event and command handler declarations.

   ```
   OSStatus RegisterMoonTravelHelp (void);
   ```

3. Create the function `RegisterMoonTravelHelp`.

   Copy the following code to the bottom of the `main.c` file:

```
OSStatus RegisterMoonTravelHelp (void)
{
  CFBundleRef    myAppsBundle;
  CFURLRef       myBundleURL;
  FSRef          myBundleRef;
  OSStatus       err = 0;

  myAppsBundle = NULL;
  myBundleURL = NULL;

  myBundleURL = NULL;

  myAppsBundle = CFBundleGetMainBundle();                          // a
  myBundleURL = CFBundleCopyBundleURL(myAppsBundle);               // b
  CFURLGetFSRef (myBundleURL, &myBundleRef);                       // c
  err = AHRegisterHelpBook (&myBundleRef);                         // d
  CFRelease (myBundleURL);                                         // e
 return err;
}
```

a. `CFBundleGetMainBundle` returns a reference to the application's main bundle.

b. `CFBundleCopyBundleURL` returns the location of the bundle.

c. `CFURLGetFSRef` returns the `FSRef` for the bundle URL.

d. `AHRegisterHelpBook` registers the help book associated with the application's file specification reference (`FSRef`).

e. `CFRelease` releases the memory associated with `myBundleURL`.

4. Add code to register the Moon Travel Help book.

   Type the following just before the statement `RunApplicationEventLoop`:

   ```
   RegisterMoonTravelHelp();
   ```

Build, Run, and Test the Application

The Moon Travel application should be complete. Follow the instructions in "Build, Run, and Test the Application" (page 82).

- After the application launches, choose Moon Travel Help from the Help menu. You should see the following page: