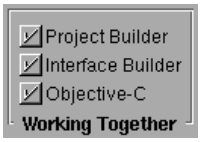


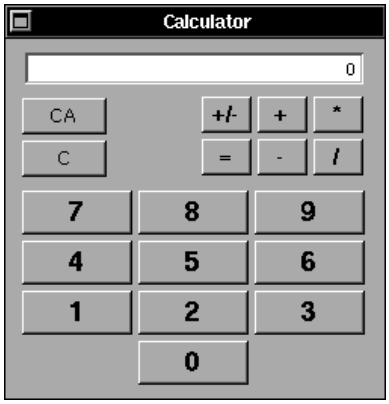
5

by Simson L. Garfinkel and Michael K. Mahoney

Building a Project: A Four Function Calculator

Copyright ©1992 Springer-Verlag Publishers

In this chapter we'll build a simple calculator application with four functions: add, subtract, multiply, and divide. When we're done, our calculator will contain the menu and window shown below. In the process of building the calculator, we'll learn a lot more about Interface Builder, connections, and some of the commonly-used NeXTSTEP Application Kit (AppKit) classes.



We've chosen to build a calculator as the first "real" application in this text for several reasons. First of all, calculators are familiar. We've all used one, and we sort of know how they work. (When creating an application, the first thing to understand is the problem to solve.) Second, calculators are useful. As programmers, we're constantly having to do silly little things like add two numbers together or convert a number from decimal to hexadecimal (the hex part will be built in the next chapter). It's a tool that you can put to work after you build it.

More importantly, a calculator is a good starting-off point for budding NeXTSTEP developers. In subsequent chapters, we'll use the calculator as a building block for learning about NeXTSTEP graphics, printing, multiple windows, file handling, and many other features.

Creating your own calculator puts you in charge of its design. After all, there are so many different kinds of calculators: some are scientific, some financial, and some are just simple four function calculators. Our calculator will let you key in the sequence "3+4=" by clicking four buttons in a window and will display (in order) 3, 3, 4, and 7 in a text output area. If you don't like the decisions we've made and want to change or add functions and features, go right ahead! Our hope is to give you the know-how to create your own applications.

Getting Started Building the Calculator Project

The six steps immediately below apply only to NeXTSTEP 3.0. If you are using NeXTSTEP 2.1, then you create a project in a much different way than in NeXTSTEP 3.0, and you should skip to "Building a Project in NeXTSTEP 2.1" on page 124.

1. Make sure the Project Builder (PB) and Interface Builder (IB) icons are in your dock and then launch PB from your dock. All you see is the main menu. See Figure 1 below.
2. Choose PB's **Project→New** menu command (or type Command-n) to create a new application. The **New Project** panel opens up. See Figure 2 below.
3. Type "**Calculator**" in the **Name** field of the **New Project** panel as in Figure 2 and then hit the Return (or Enter) key. A main window for the Calculator project opens. See Figure 3 below.

FIGURE 1. Main and Project Menus in Project Builder

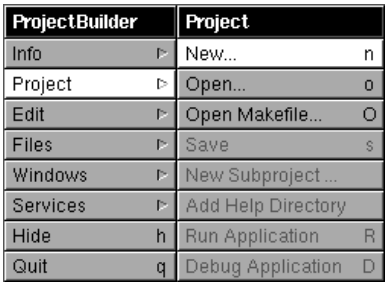
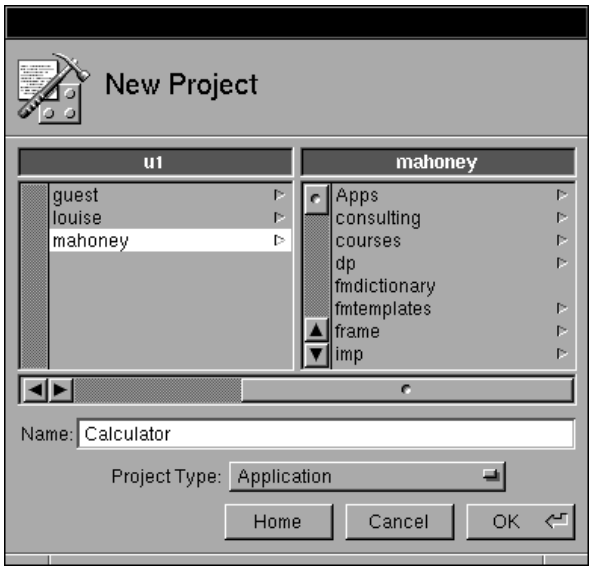


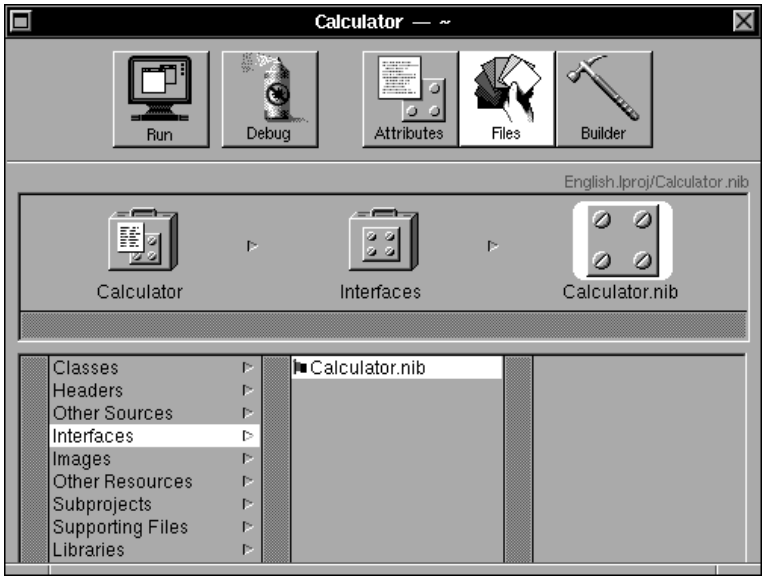
FIGURE 2. New Project Panel in Project Builder



The main window in Project Builder contains five buttons just below its title bar. The two buttons on the left, **Run** and **Debug**, are "action" buttons that run and debug your project. The three buttons on the right, **Attributes**, **Files**, and **Builder**, select the content (or view) of what you see in the window, much like the buttons at the top of Interface Builder's Palettes window. We'll discuss these buttons in more detail later.

When you first see PB's main window the Files view is displayed. While in the Files view, the middle part of the main window contains an icon path similar that in a Workspace File Viewer. Below the icon path is a browser

FIGURE 3. Main Window in Project Builder



which contains a list of all the file types associated with a project. We'll discuss these different file types in "The Files in a Project" on page 159.

4. Click **Interfaces** in the main window's browser and then click **Calculator.nib** under **Interfaces**. The **Calculator.nib** file icon appears in the icon path as in Figure 3.

5. Double-click the **Calculator.nib** file icon in the main window's icon path. Interface Builder (IB) will automatically launch and display the **Calculator.nib** interface provided by PB, which includes a main menu titled "Calculator" and a main window titled "My Window." An associated File window is also displayed in the lower left corner of the screen.

6. In order to simplify the screen, click in Project Builder's main window and then type Command-h to hide Project Builder.

Skip the next section, which only applies to NeXTSTEP 2.1, and go directly to "Building the Calculator's User Interface" on page 126.

Building a Project in NeXTSTEP 2.1

1. Make sure the Interface Builder (IB) icon is in your dock and launch IB from your dock.

2. Choose IB's **File→New Application** menu command (or type Command-n) to create a new application. IB automatically provides an interface which includes a main menu titled "UNTITLED" and a main window titled "My Window." An associated File window containing objects and resources is also displayed in the lower left corner of the screen.

3. Choose IB's **File→Save** menu command (or type Command-s) to save this interface. When the **Save** panel pops up, enter (without the quotes or any spaces) "**Calculator/Calculator**" as the directory and file in which to save the application.

4. Since the directory that you specified (**Calculator/**) doesn't exist, IB will ask you if you want to create a new "path." (Actually, this is a standard feature of the NeXTSTEP Save panel object.) Click **Create**.

When you click **Create**, Interface Builder creates two things: a directory named **Calculator,** and a file named **Calculator.nib** in that directory. IB also changes the menu's title to "Calculator."

5. Choose IB's **File→Project** menu command to create a project.

In the lower-right hand corner of the screen, you'll see Interface Builder's Project Inspector, which is one of the views in IB's Inspector. There are four kinds of projects that IB can create. The **Application** choice is what you usually use; it tells IB that you're creating a project that's going to be compiled into an application program. The **Custom IB** and **IB Palette** choices are used for creating your own version of IB and your own IB palette, respectively. Your own palette can be added to IB and used just like the palettes delivered by NeXT. There are also a growing number of third-party commercial and public-domain palettes available for NeXTSTEP.

Subproject is used for managing projects that are too complicated to put everything in a single directory. This option lets you put nibs and source code into their own directories, which get separately compiled and linked into the main program.

6. Leave the selection on **Application** and click **OK** in IB's Project Inspector. This will create a NeXTSTEP 2.1 project consisting of a Makefile and three other supporting files in the **Calculator** directory. A fifth project file, **Calculator.nib** was already saved in the directory.

Now you'll see the "real" Project Inspector which lets you monitor all of the files that are part of your project. The **Add** button allows you to add files to your project, while the **Remove** button allows you to remove them.

If you want to edit a file, simply click the file's name in the right-hand column and click the **Open** button (alternatively, you can double-click the file's name). If the file is a nib file, Interface Builder will open up that nib's File window and display the nib's objects. Otherwise, Interface Builder will send a message to Workspace Manager asking it to open the file. This is very handy when you are creating class files, because you can go directly from IB to editing the class files in the Edit application.

Building the Calculator's User Interface

The **Calculator.nib** file created above is called, aptly enough, a *nib* file (*nib* stands for NeXT Interface Builder.) A nib file stores information about all of the user interface objects in your program – the windows, controls, menus, etc. and the connections between them – and the other objects that IB "knows about." When you compile and link your program, the program's nib file (or nib files, if the program uses more than one) gets bundled together with the program's executable code and stored a *package* or *app wrapper* directory. This directory has an **.app** extension and looks like an executable application in the Workspace File Viewer.[1]

The nib files are stored in a NeXTSTEP proprietary binary format that is undocumented. Fortunately, it doesn't need to be documented – all of the management of the nib files is done by IB. IB is a nib editor: when it opens a nib file, it reads the specifications and displays the associated objects. After you make your modifications to the program, IB writes out a new nib file, replacing the old one.

Now that we've created the project we'll add and customize the windows, panels and menus needed for the Calculator's user interface.

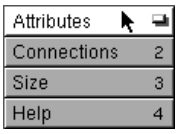
Customizing the Main Window

The main window in the Calculator's interface, currently called "My Window," doesn't look anything like a calculator: it's the wrong shape, it shouldn't have a resize bar, and it doesn't even have the right name! Fortunately, these are all properties that we can easily change by using IB's Window Inspector.

---

1. In NeXTSTEP 2.1, nibs can also be stored directly in a *section* in the executable file. See Chapter 9 for more on sections.

To see the Window Inspector for a particular window, the window must be selected. You can select a window by simply clicking in its background, or by clicking its icon in the File window's Objects view. If you click an object (e.g., button) inside a window object in IB, then the button, not the window, will be selected.



In general, the title and contents of IB's Inspector panel change in response to which object in the interface is selected. When the Inspector changes in response to a selection, then you may still have to choose which aspect of the object you want to inspect: its attributes, connections, or something else. You can make this choice by dragging to it in the Inspector's pop-up list or by typing Command-1 for **Attributes**, Command-2 for **Connections**, and so forth.

Now we'll go through the steps to customize our Calculator's window.[1]

7. Select "My Window" in IB by clicking in the window's background.
8. If necessary, press the pop-up list button in the Window Inspector panel and drag to **Attributes** (or type Command-1). The Window Inspector should now look like the one on the left of Figure 4 below.
9. Change the **Title** from "My Window" to "Calculator" and hit Return.
10. Turn off the **Close** and **Resize Bar** attributes in the Window Inspector by clicking their switches so the check marks disappear; see the Inspector on the right of Figure 4. (In NeXTSTEP 2.1, you'll have to click the **OK** button to make the changes take effect.)

The resize bar in the now-renamed Calculator window will disappear. The close button will *not* disappear until the interface is tested or the application is running. This is so you can close the window to simplify the screen while building the interface.

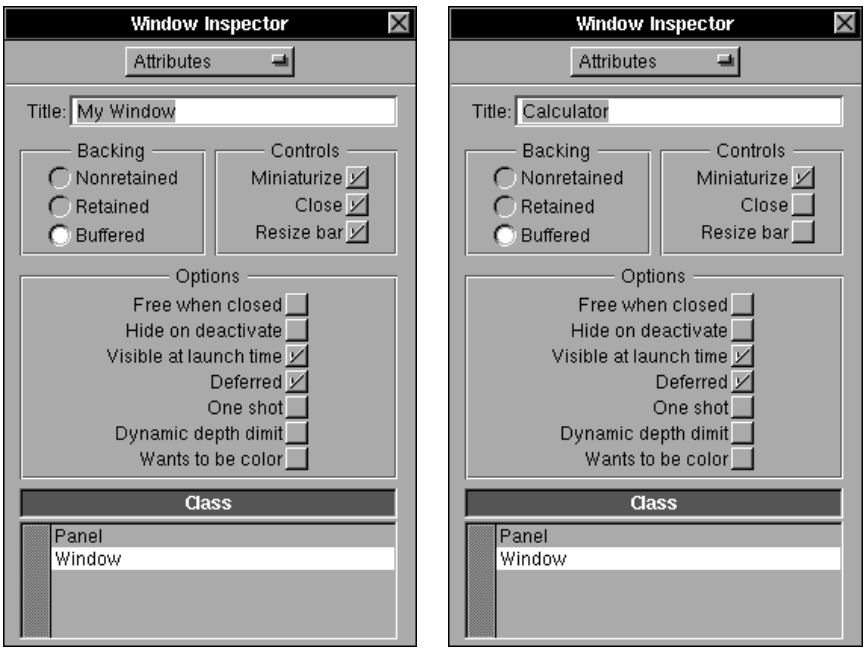
Now resize the window to make it into the shape of a calculator.

Wait a minute! How do you resize the window if you just made the resize bar go away? Well, that's the purpose of the funny icon ( ) in the upper-left hand corner of the Calculator window. We'll refer to this as the *resize button*.

---

1. These steps work in both NeXTSTEP 3.0 and 2.1; however, the screen shots are all taken from NeXTSTEP 3.0 and may differ slightly in 2.1. Where there are significant differences between 2.1 and 3.0, we'll mention them in footnotes.

FIGURE 4. Window Inspector, Before and After



11. Click the resize *button* and the resize *bar* will temporarily reappear at the bottom of the Calculator window.

12. Resize the Calculator window so that it is about 3 inches square. After you resize the window, the resize bar will disappear again, so you might have to click the resize button several times if you wish to try several sizes.

Adding Controls in a Window

Next we'll drag the buttons and text display area that the Calculator application will need from Interface Builder's Palettes window into the **Calculator** window.

13. Make sure the Views palette is visible by clicking the Views button at the top of IB's Palettes window.

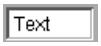
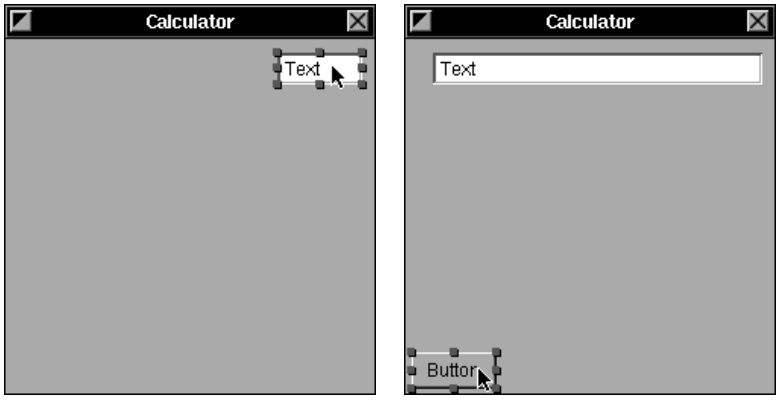
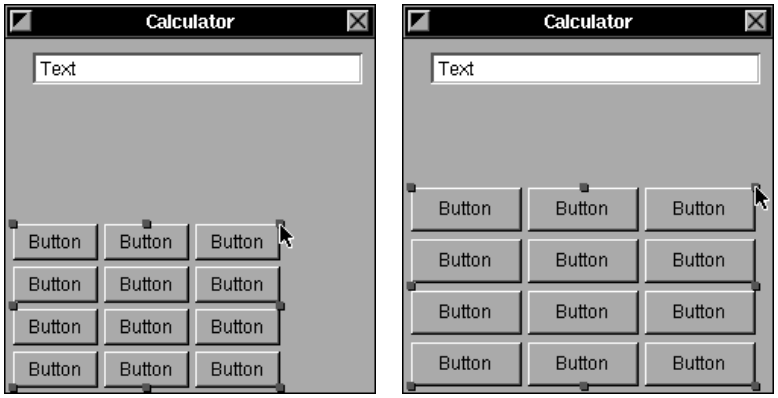
14. Drag a **TextField** () object from the Palettes window and drop it near the top left of the Calculator window. The window should look like the one on the left of Figure 5 below.

FIGURE 5. Adding a TextField and a Button to a Window



15. Drag the left-middle handle to the left to widen the **TextField** so that it is almost the width of the Calculator window, as in the window on the right of Figure 5.
16. Drag a **Button** (Button) object from the Palettes window and drop it in the lower left corner of the Calculator window as in Figure 5.
17. While holding the Alternate key on your keyboard, drag the upper-right-hand handle of the button up and to the right. Release the mouse button when there are four rows by three columns of buttons, as in the window on the left of Figure 6 below. Congratulations! You've just created a *matrix* of buttons.

FIGURE 6. Adding a Matrix of Buttons to a Window



Matrix is one of the classes provided by the NeXTSTEP Application Kit. A **Matrix** object is a 2-dimensional array of other objects that are subclasses of the NeXTSTEP **Cell** class.

Any **Cell** object can be displayed in a **Matrix**. Interface Builder automatically converts a **Control** and its associated **Cell** object into a **Matrix** and a group of **Cell** objects when you drag one of the **Control**'s resizing handles with the Alternate key pressed.

Every NeXTSTEP **Control** class, including **Button**, **Slider**, and **TextField**, has an associated **Cell** subclass (for example, **ButtonCell**, **SliderCell**, and **TextFieldCell**). These cell objects do the actual drawing of the controls that we put into the window.

NeXTSTEP uses combinations of objects because it is faster to draw in a **Cell** than in a **View**; there is less overhead. The corresponding **View** subclasses, **Button**, **Slider**, and **TextField**, are still used for handling events. Interface Builder hides this split personality between the **Control** and **Cell** from us and makes the control and its associated cell look like a single object. This is often a source of confusion for programmers new to NeXTSTEP.

18. *Without* pressing any modifier key, drag the upper-right handle of the **Button** matrix up and to the right. This time, the buttons will get bigger as in the Calculator window on the right of Figure 6.

Matrix Dragging Options

When you drag a handle on a matrix object, one of three things can happen depending on which modifier key is pressed. We saw the first two of these in the example above.

Modifier Key	Effect on the Cells in the Matrix
none	Changes the size of all cells in the matrix.
Alternate	Changes the number of cells in the matrix.
Command	Changes the spacing between cells.

The buttons in the **Matrix** we created above will be used to represent digit keys on our Calculator, and thus we'll change their names from "Button" to the 10 decimal digits (and disable the remaining two). We also need to set some less obvious attributes of the buttons, called *tags*, to make the buttons work properly. In order to explain how tags work and better understand why we make certain choices while creating an interface, we'll postpone finishing the interface to discuss the Objective-C class that we'll create to handle the button clicks.

Building the Calculator's Controller Class

It's time to start thinking about the Objective-C object that will control our Calculator – that is, respond to button clicks, calculate the values that the user wants, and display the results. By convention, this kind of object, which performs behind-the-scenes work and communicates with the user interface, is called a *controller.*

Controllers generally don't have main event loops; instead, they perform actions in response to events that are received and interpreted by other objects. A good rule of thumb is to place as little code in your controller as necessary. If it is possible to create a second controller that is only used for a particular function, do so – the less complicated you make your application's objects, the easier they are to debug. Our Calculator's controller will contain the code to perform the arithmetic and thus can be thought of as the *computational engine* or *back end* of the application.

Designing the Controller Class

NeXTSTEP doesn't provide you with a **Controller** class – it's up to you to write one for your particular application. (IB and the AppKit are fabulous but they can't do *everything* for you – at least not yet!)

Before you start coding, it's a good idea to sit down and think about your problem. What does the controller have to do? What sort of messages will it need to respond to? What sort of internal state does it have to keep in order to perform those functions? Recall that our Calculator will let a user key in the sequence "2*5=" by clicking four buttons in a window and will display (in order) 2, 2, 5, and 10 in a text output area. Thus for our Calculator, the answers are fairly straightforward.

What our Calculator must do:

- Clear the display and all internal registers (value holders) when a *clear* button is clicked.
- Allow the user to click a digit button on the numeric keypad and display the corresponding digit immediately after it is typed.
- Allow the user to click a function button (e.g., add, subtract).
- Clear the display when the user starts entering a second number.
- Perform the appropriate arithmetic operation when the user presses the "equals" button or another function button.

Our Calculator must also maintain the following state to perform these functions:

- The first number entered.
- The function button clicked.
- The second number entered.

It turns out that, in order to work properly, our controller object needs two more pieces of information:

- A flag that indicates when a function button has been clicked. If the flag is set, then the text display area (which we'll call **readout**) should be cleared the next time that a digit button is clicked, because the user is entering a second number.
- The location in the **readout** text display area where the numbers should be displayed.

If you think about it, what we are actually doing is using Objective-C to create a simulation of a real, physical calculator. That's what object-oriented programming is often about: constructing progressively better simulations of physical objects inside the computer's memory, and then running them to get real work done. When the simulation is functionally indistinguishable from the real-life object being simulated, the job is finished.

Creating the Controller Class

Every Objective-C class, except **Object**, is based on another class. The **Object** class itself is the most fundamental Objective-C class, because it defines the basic behavior of all objects and is at the root of all inheritance hierarchies. Since we don't need any other special behavior in our Calculator that is already defined in the AppKit, our **Controller** class will be a subclass of **Object**.

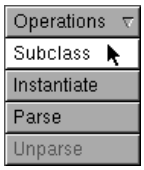
We'll start building our **Controller** class by subclassing it from **Object** in Interface Builder.

1. Open the Classes view in IB's File window by clicking the Classes (**h**) suitcase icon.[1]

1. In NeXTSTEP 2.1, *double*-click the **Classes** suitcase icon in the File window. This will open up a *separate* Classes window.

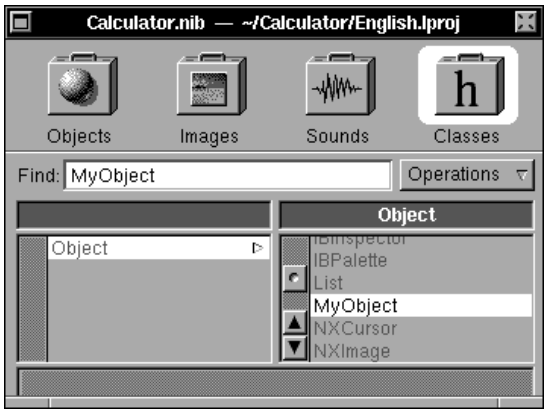
2. Scroll to the far left in the Classes browser by clicking the left arrow (◀) several times, and then select the **Object** class by clicking it. See Figure 7 below.

The **Object** class name is displayed in gray, which means that you can't change any of its properties or built-in behaviors without subclassing it. So that's what we'll have to do.



3. Drag to **Subclass** in the **Operations** pull-down list. A new class called **MyObject** will appear under **Object** in the class hierarchy as in Figure 7 below.

FIGURE 7. Classes View in Resource Window



4. Double-click the **MyObject** class name in the white text field area in the Class Inspector (at the lower right of the screen), change the name from **MyObject** to **Controller**, and hit Return.

You've just created a new class called **Controller**. Right now it doesn't do anything different than the **Object** class. Next we'll give the **Controller** class some custom behavior by adding some "outlets" and "actions."

Outlets and Connections

NeXTSTEP uses a powerful system called *outlets* and *connections* to give you an easy way to send messages between user interface objects such as windows, buttons, other controls, and your own custom objects. An *outlet* is simply an instance variable in an Objective-C class that has the type **id**, and thus can store a pointer to an object. The value of this instance variable is usually the **id** of another object in the nib, i.e., a user interface object.

When an outlet is set to store the **id** of another object in the nib file, Interface Builder calls this a *connection.* NeXTSTEP automatically maintains connections for you. When object specifications are saved in a nib file, the connections we set up between them in IB are saved as well. These connections are automatically restored when the nib file is loaded back into IB.

For example, suppose that you have two object specifications in a nib file: object A and object B. Suppose also that object A contains an outlet that points to object B. When NeXTSTEP loads this nib file, it will first create new instances of object A and object B. NeXTSTEP will then automatically set the outlet in object A to point at object B. That is, it sets the outlet A to be the **id** of object B.

Outlets therefore give you an easy way to track down the **id** of objects that are dynamically loaded with nib files! They are the mechanism that NeXTSTEP gives you to "wire up" an interface without writing any code.

Adding Outlets to an Object

There are two ways to *add* outlets to a class. You can either add outlets by entering them in Interface Builder's Class Inspector, or you can add outlets "by hand" using an editor to type them into the class interface (**.h**) file for your class. In the latter case you have to tell IB to **Parse** the interface file so IB knows that the outlets exist for that class. We'll see how to add outlets in IB in this chapter and "by hand" in the next chapter.

Making the Connection

After adding an outlet, you use IB to *initialize* where it points. You do this by setting up a connection from the object containing the outlet to the object you want it to point to, and then choosing the outlet from the list of outlets in IB's Connections Inspector. When you make a connection between an outlet in an object and another object in IB, all IB does is set the instance variable in the first object to the **id** of the object to which it is connected. That's all!

In the steps below we'll add and initialize an outlet in IB.

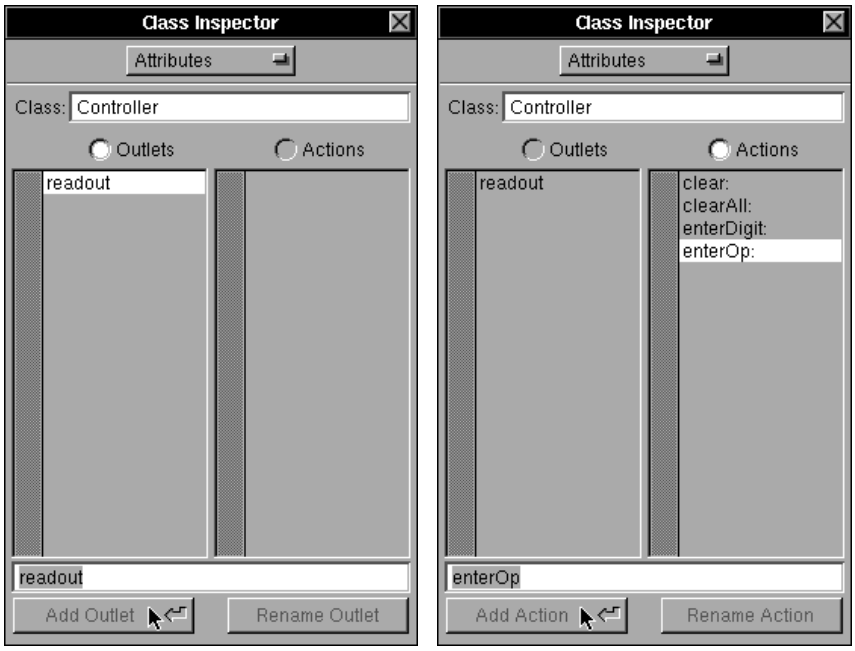
5. If necessary, select **Controller** in IB's File window and type Command-1 to display IB's Class Attributes Inspector.

6. If necessary, click the **Outlets** radio button in IB's Class Inspector. This tells IB that you want to edit the outlets in the **Controller** class. See Figure 8 below.[1]

7. Add an outlet called **readout**. Do this by clicking in the white text area near the bottom of the Class Inspector, typing "**readout**," and then clicking the **Add** button.

   If you make a mistake, you can use the **Rename** button to change the name of an outlet. You can also use the **Edit→Cut** menu command to remove an outlet. When you're done, the Class Inspector should look like the one on the left in Figure 8 below.

FIGURE 8. Adding Outlets and Actions in the Class Inspector



We'll use this **Controller** outlet to point to the text display area (**TextField**) object, so the **Controller** can send it messages. Next, we'll add action methods to the **Controller** class.

1. In NeXTSTEP 2.1, make sure the word **Outlet** is highlighted in black on the

button in the Class Inspector (as it is here).

Adding Actions to the Controller

An "action" is a special type of Objective-C method. Action methods are special because they take a single argument called **sender**, the **id** of the object that sent the message which invoked the action method. Using Interface Builder, we can arrange for an object's action method to be automatically invoked in response to a user event, such as a button click, menu choice, or slider drag. Thus an action method is an "event handler."

In Chapter 3, the **takeDoubleValueFrom:** action method was used to make a **TextField** automatically take its value from the **Slider** object when the slider knob was moved (see Figure 9, "Communication between Slider and TextField," on page 85). Here we're going to create our own action methods in the **Controller** class and arrange to have them invoked when the user clicks our Calculator's buttons.

8. Click the **Actions** radio button in IB's Class Inspector. This tells IB that you want to edit the actions in the **Controller** class. See Figure 8 above.[1]

9. Add the following four actions to your **Controller** class (you don't have to type the colons ("**:**") as IB will automatically append them):

```
clear:
clearAll:
enterDigit:
enterOp:
```

In light of our discussion of the design of the **Controller** class, the function of these four actions should seem fairly self-evident. The **Controller** Class Inspector should look like the one on the right in Figure 8 above.

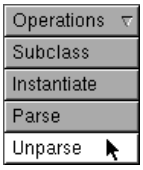
Notice that there's only one action to handle all digit button clicks (**enterDigit:**) and only one action to handle all the function buttons (**enterOp:**). The way we determine which digit or function button is clicked is to use the single argument of these actions, the **id** of the *sender* of the message. By querying the sender of the message, the methods **enterDigit:** and **enterOp:** find out which digit or which function button was clicked and can then perform the appropriate action. This is a much more economic means of method dispatch than creating a separate method for each button on our Calculator – it takes less code but it runs virtually just as fast.

---

1. In NeXTSTEP 2.1, make sure the word **Action** is highlighted in black on the Actions-Outlets button in the Class Inspector.

Generating the Controller Class Files

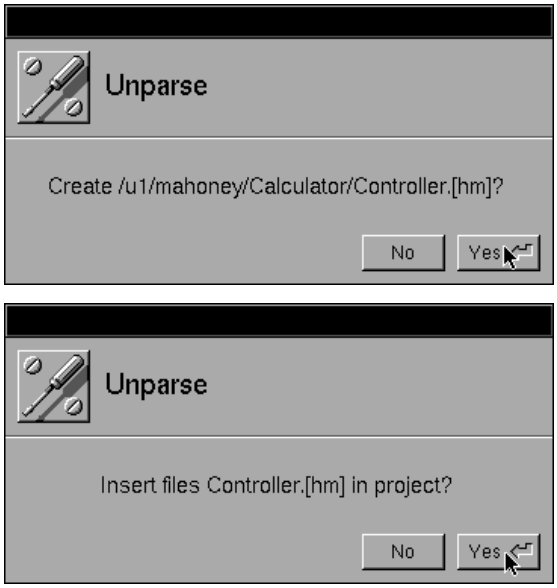
We need to tell Interface Builder to create the **Calculator.h** *interface* and the **Calculator.m** *implementation* class files, so we can add the appropriate functionality and eventually compile them with the Objective-C compiler. Interface Builder's **Unparse** command generates these files from the class specifications we made in the File window and Class Inspector.



10. Make sure the **Controller** class is selected in the File window (Classes view) and then choose the **Unparse** operation from the **Operations** pull-down list.

11. Since creating these new files would obliterate any files with the same name that already exist, IB asks us to confirm the operation. (See the attention panel at the top of Figure 9 below.) Click **Yes**.

FIGURE 9. Unparse Attention Panels in Interface Builder

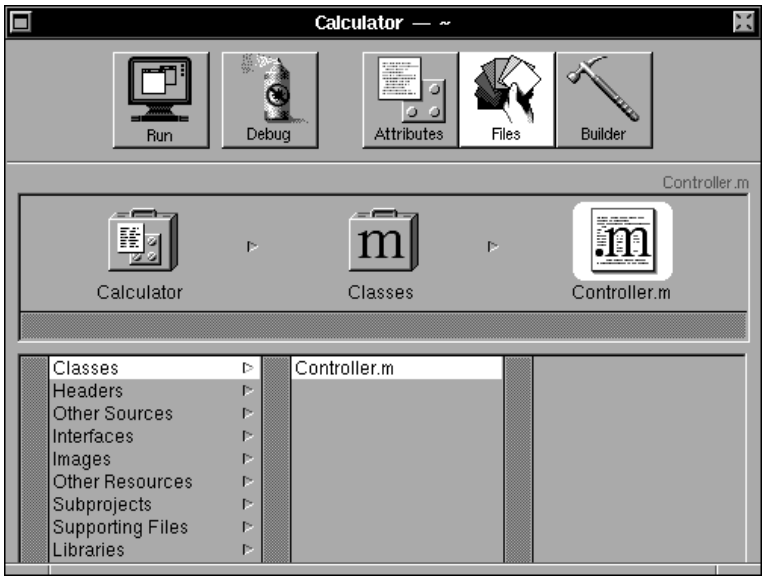


12. IB next asks if these files should be inserted into the project. (See the attention panel at the bottom of Figure 9.) Click **Yes**.

After you complete the last step above, the Project Builder tool reappears and displays the new **Controller** class implementation file, **Controller.m**, in the Files view as in Figure 10 below. The **Controller.h** class *interface* file appears under **Headers** in PB's Files view. These new **Controller** class files reside in the **~/Calculator** directory and contain only a skeleton of the

Controller class. In order to make our **Controller** work, we've got to write some Objective-C code.[1]

FIGURE 10. Controller Class in Project Builder



Adding Code to Make the Controller Class Work

In order to make the **Controller** work, we need to understand a little bit about four-function calculators. The basic four-function calculator has three registers: a X and a Y register, both of which hold numbers, and an operations register, which holds the current operation. The screen always displays the contents of the X register. Clicking a function button stores that function in the operation register and sets a flag. If the flag is set, then the next time a digit button is clicked, the number in the X register is moved to the Y register and the X register is set to 0.

We're going to get the **Controller** class working in stages, testing them one at a time. Generally, this is a good approach to writing any program, large or small. Object-oriented programming makes it easy to test the individual parts, because they are all fairly self-contained.

---

1. In NeXTSTEP 2.1, the Project Inspector reappears and displays the class files. Most of the information and functions available in IB's Project Inspector in NeXTSTEP 2.1 have been moved to Project Builder in NeXTSTEP 3.0.

First, we'll get numeric entry and the clear keys working. Later we'll handle the arithmetic functions.

13. Open the **Controller.h** class interface file in your favorite editor.

You can open the **Controller.h** file in the Edit application by double-clicking **Controller** in IB's File Window or by double-clicking **Controller.h** under **Interfaces** in PB's Files view.

Below we list the **Controller.h** file containing all the lines generated by Interface Builder in `medium type` and all the lines you need to insert in **`bold type`** (we'll use this type convention throughout the book when we mix "old" code with "new" code to be inserted).

```
/* Controller.h */

#import <appkit/appkit.h>

@interface Controller:Object
{
  id     readout;
  BOOL   enterFlag;
  BOOL   yFlag;
  int    operation;
  float  X;
  float  Y;
}

- clear:sender;
- clearAll:sender;
- enterDigit:sender;
- enterOp:sender;
- displayX;

@end
```

Interface Builder generated the first two (non-bold) lines because we subclassed **Object** to create the **Controller** class (**appkit.h** will import the **Object.h** class interface file)[1]. Since we added **readout** as an outlet in the Class Inspector, IB also generated the **id** declaration for **readout**. Finally, IB generated the four action method declarations because we added the four actions in IB's Class Inspector. Note the single argument **sender** for all of these action methods.

1. In NeXTSTEP 2.1, the **Object.h** class file is imported explicitly and **appkit.h** is *not* imported. We'll discuss this further on page 159.

14. Add the five new instance variables and one new method indicated by the lines that are shown in **bold** type in the **Controller.h** file above. We'll discuss the new **displayX** "non-action" method a bit later.

15. If necessary, open the **Controller.m** file in an editor. You can do this by double-clicking **Controller.m** under **Classes** in the PB's Files view.

Below we list the **Controller.m** file. As with the **Controller.h** file above, we list the lines generated by IB in medium type and the lines you need to insert in **bold type**.

```
#import "Controller.h"

@implementation Controller

- clear:sender
{
  X = 0.0;
  [self displayX];
  return self;
}

- clearAll:sender
{
  X = 0.0;
  Y = 0.0;
  yFlag = 0;
  enterFlag = 0;
  [self displayX];
  return self;
}

- enterDigit:sender
{
  if (enterFlag) {
    Y = X;
    X = 0.0;
    enterFlag = NO;
  }

  X = (X*10.0) + [ [sender selectedCell] tag];
  [self displayX];
  return self;
}

- enterOp:sender
{
    return self;
```

```
}

- displayX
{
  char buf[256];

  sprintf( buf, "%15.10g", X );
  [readout setStringValue: buf];
  return self;
}

@end
```

Interface Builder generated the line which imports **Controller.h** because every class implementation file must import its own interface file. Most of the other lines generated by IB are simply stubs for the action methods that we set up in the IB's Class Inspector. The lines IB generates in class files are more for convenience than anything else.

16. Insert the code shown in **bold** type above into the **Controller.m** file.

Note that the **Controller** sends messages to instances of the **TextFieldCell** and **Matrix** classes. For example, the newly added **displayX** method displays the contents of the X register by sending the **setStringValue:** message to **readout**, the outlet which we'll initialize in IB to point to the **TextFieldCell** object (near the top of the Calculator window).

When a message is sent to an object of a class, then the class interface file definition for that class should be **#import**-ed in the class definition. But **#import** statements for the **TextFieldCell** and **Matrix** class interface file definitions are not listed in the code above, so what's going on? Fortunately, the **#import "Controller.h"** line in **Controller.m** together with the **#import <appkit/appkit.h>** line in **Controller.h** take care of importing the **TextFieldCell** and **Matrix** class interface file definitions for us.[1] In fact, they import *all* Application Kit class definitions. This is very ineffi-

---

1. If you're using NeXTSTEP 2.1, you'll have to insert the
   #import <appkit/appkit.h>
line yourself. Alternatively, you can include the **TextFieldCell** and **Matrix** class interface files explicitly for improved compilation efficiency. To do this, you would insert the following two lines at the top of either **Controller.h** or **Controller.m**:
   #import <appkit/TextFieldCell.h>
   #import <appkit/Matrix.h>

cient in NeXTSTEP 2.1 because compilation time is greatly increased. However, in NeXTSTEP 3.0 the AppKit class headers are all precompiled so it's okay to import them all. (A *precompiled header* file has been preprocessed and parsed, thereby improving compile time and reducing symbol table size.) This is why IB in NeXTSTEP 3.0 inserts the **#import <appkit/appkit.h>** line in all class interface files it generates.

The **clearAll:** method in the **Controller.m** file sets the X and Y registers to **0.0** and the two flags to false, and then sends the **displayX** message to **self** (the **Controller** object itself) to display **0.0** in the text display area. The **clear:** method is similar but only needs to set the X register to **0.0** and redisplay. We'll discuss the **enterDigit:** and **enterOp:** methods after we finish setting up the user interface and making all the connections.

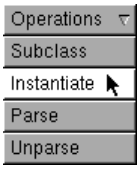
Customizing Buttons and Making Connections

In this section we'll use Interface Builder to add more interface specifications to the **Calculator.nib** file, including customizing buttons and making several different types of connections between objects. In order to make connections which involve an object of the **Controller** class, we need a representation of it in IB.

Instantiating (Creating an Instance of) the Controller Class

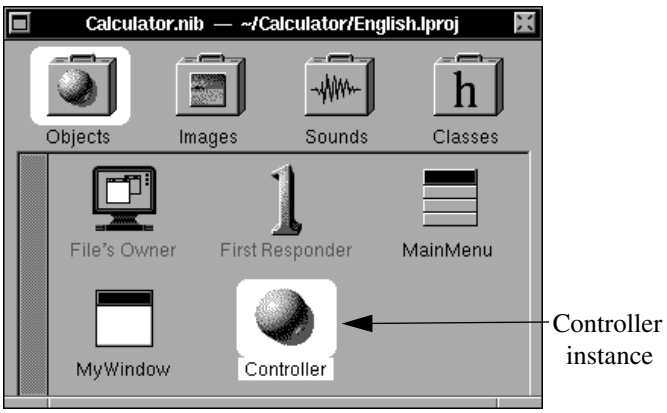
Creating the *class* isn't enough: we also need to create an *object* that is a member of this class, called an *instance*. Then we have to arrange for the numeric keypad of buttons in the Calculator window to send action messages to the instance whenever these buttons are clicked.

1. Make sure the Classes view is displayed in IB's File window and then select the **Controller** class.
2. Choose the **Instantiate** command from the **Operations** pull-down list.



This will create an icon called **Controller** in the Objects view in the Calculator's File window as in Figure 11 below (IB automatically displays the Objects view). This icon represents an instance object of the **Controller** class; it can be used as the *target* object of action messages and also to initialize outlets. You can change the name **Controller** if you want; it isn't used for anything except your convenience.

FIGURE 11. Controller Instance Object in File Window



Setting Up Titles and Tags for the Keypad Buttons

Next, we'll set up the buttons on the numeric keypad for the digits 0 through 9 and arrange for them to send messages to the **Controller** object. The **Controller** object needs to differentiate between the buttons somehow, so we'll assign them different integer *tags*.

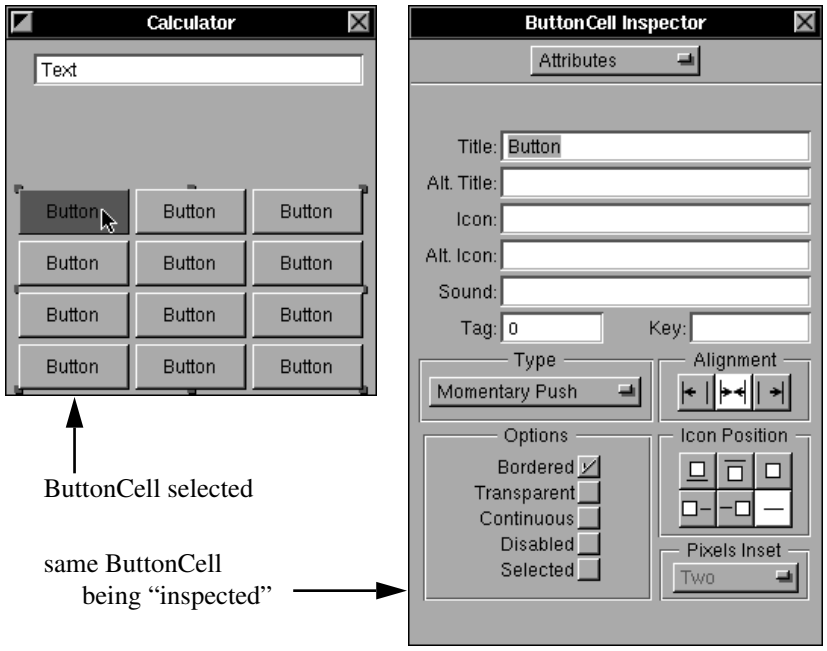
3. Double-click the button at the upper-left of the matrix in the Calculator window. The button highlights in dark gray to indicate that it is selected, and the Attributes Inspector will display the button's attributes. See Figure 12 below.
4. Change the **Title** of the button from "Button" to "7" and hit Return.
5. Change the **Tag** of the button from **0** to **7**.[1]

A *tag* is a reference integer for a **Control** object and can be set and read in IB's Inspector panel. Tags are not used by the AppKit; rather, their purpose is to allow your program to distinguish cells (objects) in a matrix (or other controls) from one another when different cells of the matrix send the same message to an object.

For example, when one of the buttons in the matrix of digit buttons is clicked, we'll arrange for the **Matrix** object to send the **enterDigit:** message to our **Controller** object. The **Controller** object needs to know which button (i.e., which digit) was clicked, and thus sends a message back to the

1. In NeXTSTEP 2.1, you must click the **OK** or hit Return to make changes in the Inspector take effect.

FIGURE 12. Inspecting a Button's Attributes



sender (**Matrix** object) to determine which of the button cells in the **Matrix** object was selected. The **Controller** can then get the tag for that cell and use it in the **enterDigit:** method as if it were a digit (since the tags will correspond to the digits on the buttons).

6. Change the titles and tags of the other keypad buttons to reflect the digits that they represent, as in Figure 13 below. The button with title 1 should have tag 1, the button with title 2 should have tag 2, etc. You can use the tab key to easily move between cells in the **Matrix**.

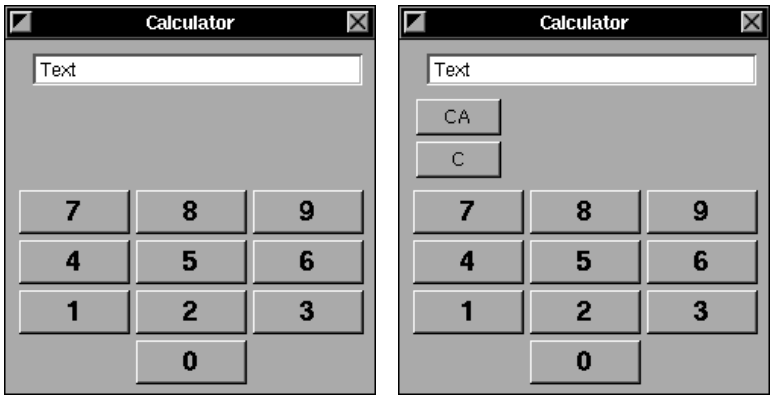
7. Double-click the lower-left hand button in the **Matrix** (which is invisible in Figure 13) to select it.

8. In the ButtonCell Inspector, *deselect* the **Bordered** switch, select the **Disabled** switch, delete the "Button" title, and hit Return. This will make the button disappear and become "unclickable."

9. Repeat Step 8 above for the lower-right hand button.

10. Click in the Calculator window background (where there are no buttons or text) to select the window, then click the button matrix *once* to select the matrix as a whole. Type Command-t to bring up the Font panel as in Figure 14 below.

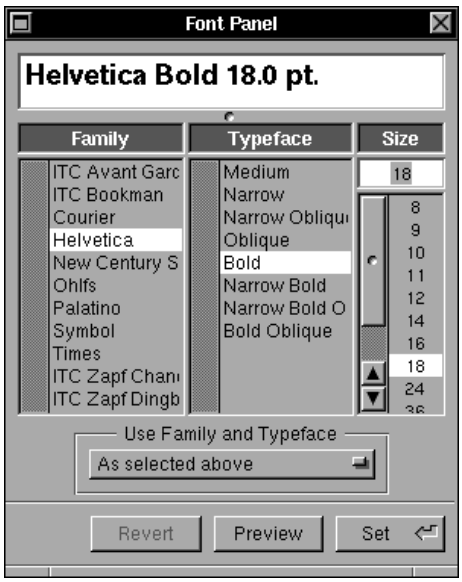
FIGURE 13. Customizing the Calculator's Buttons



As with most NeXTSTEP applications, Interface Builder lets you change the font family, typeface, and size of most text it displays. (In NeXTSTEP 3.0, a user's Preferences application settings determine the font and size of the text in the window's title bar and menus.)

11. Choose the **Helvetica Bold 18 point** font as in Figure 14 below and then click the **Set** button. (If you pick a font that's too big, the buttons and the matrix will get bigger!) Your Calculator window should look like the one on the left in Figure 13 above.

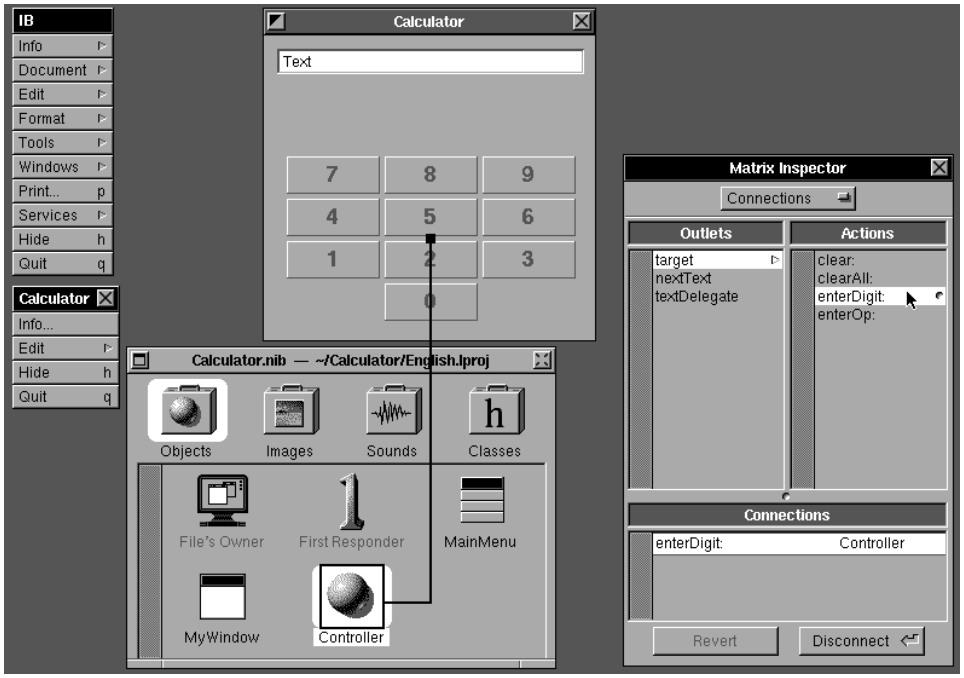
FIGURE 14. Font Panel in Interface Builder



Making the Connections

12. Connect the button **Matrix** object to the **Controller** instance object by pressing the Control key on the keyboard and dragging the mouse cursor from the middle of the **Matrix** to the **Controller** instance object icon as in Figure 15 below.

FIGURE 15. The enterDigit: Action Connection



A small black square will appear in the middle of the **Matrix** and a "connection line" will connect the **Matrix** object to the **Controller** icon. Be careful to connect the ***Matrix*** and *not* one of the individual **Matrix** *buttons*!

When you release the mouse button, Interface Builder will display the Matrix Connections Inspector. You can determine the *source* object of the connection by the name in the title bar of the Inspector – in this case it should be *Matrix* Inspector. The column labeled **Actions** in the Matrix Inspector should include the four action methods we set up earlier in the *destination* object, namely the **Controller** instance.

13. Click the **enterDigit:** action and then click the **Connect** button in the Matrix Inspector. The dimple (▪) indicates the connection was made. See the Matrix Inspector at the right of Figure 15.

The connection we just made means the following: whenever a user clicks any one of the digit buttons in the matrix, the **Matrix** object will send the **enterDigit:** action message to an instance of our **Controller** class.

WARNING: It is easy to accidentally disconnect a connection by double-clicking in the wrong place in the Connections Inspector. Be careful!

Next, we'll add *clear* and *clear all* buttons in the **Matrix** object as we did with the digit buttons. This time, however, we'll connect the buttons *individually* to the **Controller** object, not as a matrix.

14. Add a second matrix of buttons above the digits matrix in the Calculator window. This new matrix should have two buttons with titles **CA** and **C** in a font such as Helvetica Bold 16 point.

    Your window should look like the one on the right in Figure 13, "Customizing the Calculator's Buttons," on page 145. If you don't remember how to add a matrix to your window refer back to "Adding Controls in a Window" on page 128. If you can't find the Palettes window then choose in IB's **Tools→Palettes** menu command.

15. Double-click the **CA** button to select it. The ButtonCell Inspector, not the Matrix Inspector, should appear.

16. Connect the **CA** button to the **Controller** instance object icon by Control-dragging from the button to the icon and then double-clicking the **clearAll:** action in the ButtonCell Inspector (the double-click has the same effect as clicking the action name and then clicking the **Connect** button).

17. Similarly, double-click the **C** button to select it and then connect it to the **Controller** instance icon. This time, double-click on the **clear:** action.

You can make Interface Builder show you an existing connection from a source object by first selecting the source object and then clicking **target** or the action method with a dimple (▪) in the Connections Inspector.

18. Double-click the word "Text" in the **TextField** object and hit the Delete (Backspace) key to erase the word "Text."

19. If necessary, drag to Attributes in the TextField Inspector (or type Command-1).

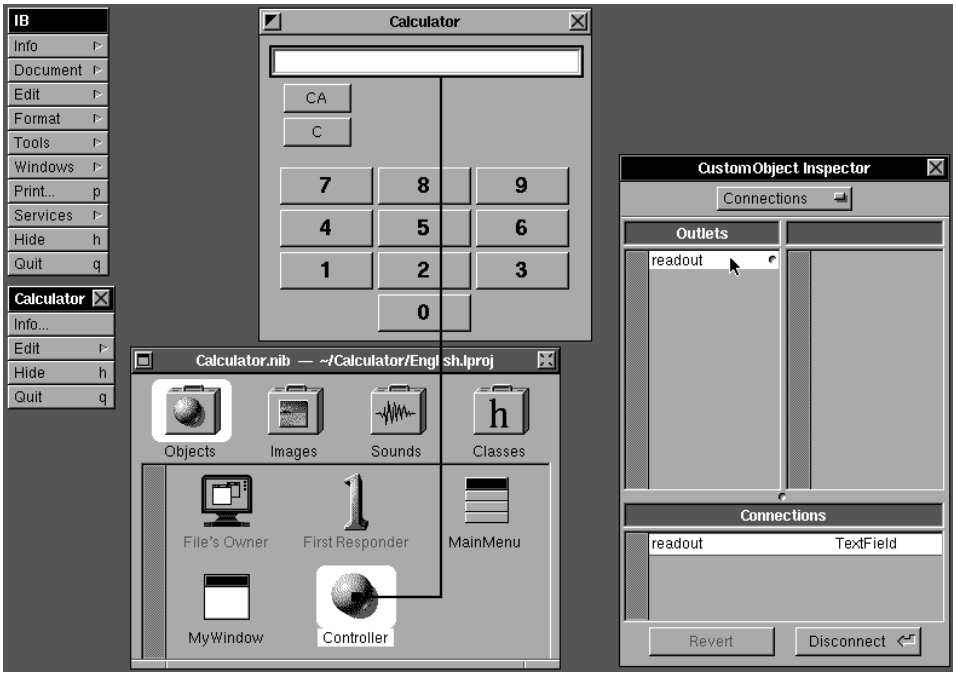
20. *Deselect* the **Editable** option in the TextField Inspector so that the text in the **TextField** object is not editable.

21. Set the alignment to be right-justified by clicking the icon that looks like this:

The source of every Calculator connection we've made so far was a user interface object, while the destination was always the **Controller** object. In the next connection we make, the direction will be reversed; the source will be the **Controller** while the destination will be an interface object. This second type of connection requires an *outlet*.

22. Control-drag from the **Controller** instance object icon to the **TextField** object. The black "connection line" should look like the one in Figure 16 below.

FIGURE 16. Making a Connection to Initialize an Outlet



23. In the Connections Inspector, double-click the **readout** outlet to complete the connection.

Connecting the **readout** outlet to the **TextField** object causes the **readout** instance variable in the **Controller** object to get initialized to the **id** of the **TextField** when the nib section gets loaded at run time. Initializing an outlet in an instance object is the only way to determine the **id** of an object created with Interface Builder, and thus outlets must be used when sending messages to user interface objects.

24. Type Command-s in IB to save the **Calculator.nib** file.

Compiling and Running a Program

At this point, we're ready to test the keypad of digit buttons. In order to do this, we must compile the **Controller.m** and **Calculator_main.m** source code and link them together with the **Calculator.nib** file. (We'll discuss the **Calculator_main.m** source file in "The Files in a Project" on page 159.)

There are three ways to compile a NeXTSTEP program: from Project Builder, from a command line prompt, or from Emacs. We describe how to do each of these below using the target **make debug**.

Compiling and Running a Program from Project Builder

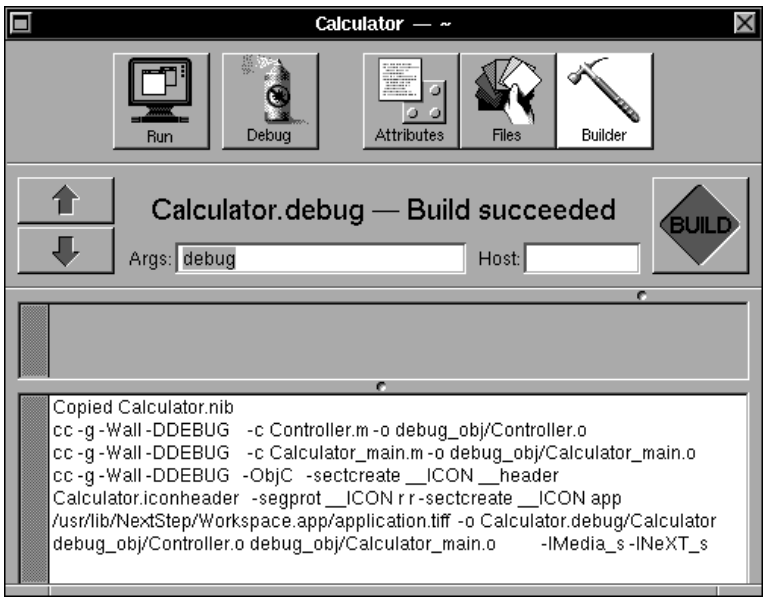
The steps below will compile (**make**, build) and run your Calculator program directly from Project Builder.

1. Activate PB and click the **Builder** button at the top of PB's main window to see the Builder view. (If you press the Command key and double-click the PB icon simultaneously, then PB will be activated and all other running applications will be hidden!)
2. In order to have PB use the **make debug** target, type "debug" in the **Args:** white text area as in Figure 17 below.
3. Start the compilation process by clicking the **Build** button in the Builder view. If there are no compile-time errors, then you should see the "Build succeeded" expression and compile log as in Figure 17. If an error occurred, see "Compiler Error Messages" on page 151 below.

4. To run your program directly from Project Builder, simply click the **Run** button at the top left of PB's main window.

   This will run the **Calculator.debug** executable file which was created in your **~/Calculator** directory when the Calculator program successfully compiled. (**Calculator.debug** is actually a *directory* containing an executable file and the **Calculator.nib** file, among others.)

FIGURE 17. Compiling the Calculator Program in Project Builder



The main Calculator window and menu appear on the screen as in Figure 18 below.

5. Try the keypad to make sure every digit works. Clicking the buttons **1**, then **2**, then **3** should make the number "**123**" appear in the white text area. The **C** and **CA** keys should zero-out the values on the display. You should see the contents of Figure 18 on the screen. Note that the default application icon is the same one as on the **Run** button in PB.

6. Choose the Calculator's **Quit** menu command to exit the program.

Compiling and Running a Program in a Terminal Shell Window

To compile your Calculator program in a Terminal shell window, you can type the **bold** text below:

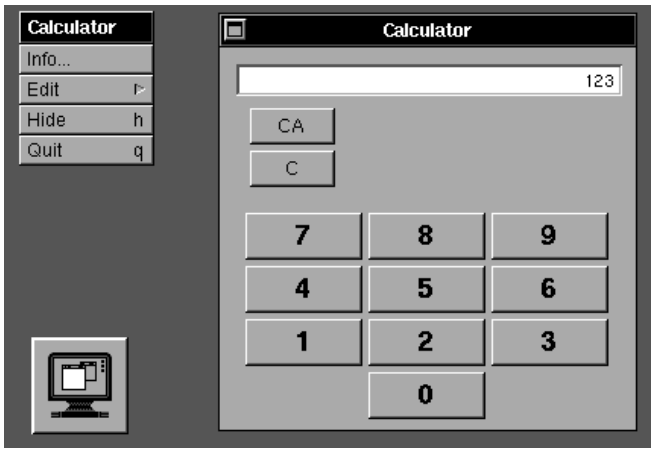
```
localhost> cd ~/Calculator
localhost> make debug
```

To compile your Calculator program within Emacs in a Terminal shell window, type:

M-x compile <*Return*>

Emacs will print:

FIGURE 18. Calculator Running in Workspace




```
make
```

Type:

```
make debug <Return>
```

The compile log should look similar to that in Figure 17 above.

If everything compiled correctly, you now have an executable file called **Calculator.debug** in your **~/Calculator** directory. You can now run your program either by double-clicking the **Calculator.debug** icon in the Workspace File Viewer or by typing the following in a shell window:

```
localhost> open Calculator.debug
```

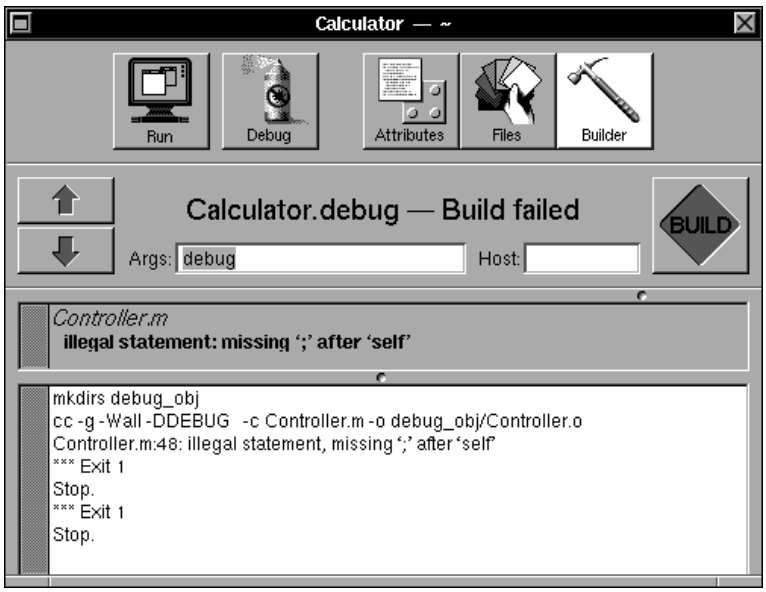
The **open** command sends a message to Workspace that it should open a file. This has the same effect as double-clicking an application icon in the Workspace Manager. See Figure 18 above.

Compiler Error Messages

Sometimes (many times!) code does not compile properly as can be seen in the Project Builder window in Figure 19 below.

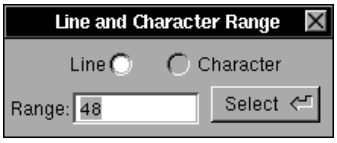
If instead of a clean compile, you get compiler warning or error messages, you have probably made a typo at some point in the code. For example, the error message in Figure 19 was generated by removing a semicolon from the **displayX** method in the **Controller.m** file.

FIGURE 19. Build Failed in Project Builder



If you get any compiler errors, we suggest that you re-examine your code line-by-line, rather than resorting to the floppy disk included with this book. Examining your code is an important skill to develop, and when you are working on your own projects, you won't have a floppy disk with the completed program to fall back on.

For the error reported here, you could open the **Controller.m** file in the Edit application, type Command-l to bring up the **Line and Character Range** panel, enter "48," the line where the error was reported, and inspect the code on line 48 and previous lines. You might also take some time to go back to Chapter 2 and review "gdb – Debugging Programs" on page 63.



If you compiled your program in a Terminal shell window, then you would get essentially the same feedback as in Figure 19 above.

Tags and the enterDigit: Action Method

The **enterDigit:** method we added to the **Controller** class is invoked whenever a digit button is clicked. Let's look at it closely.

```
- enterDigit:sender
{
  if (enterFlag) {
    Y = X;
    X = 0.0;
    enterFlag = NO;
  }

  X = (X*10.0) + [ [sender selectedCell] tag];
  [self displayX];
  return self;
}
```

The first part of the function is self explanatory: if the **enterFlag** instance variable is set, then the value of the X register is copied into the Y register and both the X register and **enterFlag** are cleared. Note that the *scope* of instance variables (e.g., **enterFlag**) is the entire class definition. All methods within a class have access to all instance variables defined in that class.

The next line contains the magic: the value in the X register is multiplied by 10 and added to the returned value **[[sender selectedCell] tag]**. This performs a base 10 left-shift operation on X and then adds the last number pressed. Let's look at this nested method expression in pieces.

[sender selectedCell] sends the **selectedCell** message to the variable **sender**. When the **enterDigit:** method is invoked (called), **sender** is set to be the **id** of object that sent the message, in this case the **Matrix** object. Clicking a button in a matrix selects that button. Thus the expression **[sender selectedCell]** returns the **id** of the **ButtonCell** object for the button that was clicked. **[[sender selectedCell] tag]** then sends the **tag** message to the **ButtonCell** object to get its tag. Since we set up the tag of each button on the keypad to be the value of that button, the expression **[[sender selectedCell] tag]** returns the numeric value of the digit that's on the button.

Adding the Four Calculator Functions

We still need to add the functions that perform the calculations to our Calculator. To do this, we'll first make some additions to the **Controller** class definition and then add the necessary on-screen function buttons.

1. Using an editor, insert the following enumerated data type after the **#import** directive in the **Controller.h** file.

```
enum {
   PLUS       = 1001,
   SUBTRACT   = 1002,
   MULTIPLY   = 1003,
   DIVIDE     = 1004,
   EQUALS     = 1005
};
```

These codes correspond to the tags that we will give the buttons in the arithmetic operations **Matrix**. The **Controller** object will determine the tag of the button that sends it the action message to decide which function button the user has clicked.

2. Using an editor, insert the lines in **bold** below into the method **enterOp:** in the **Controller.m** file.

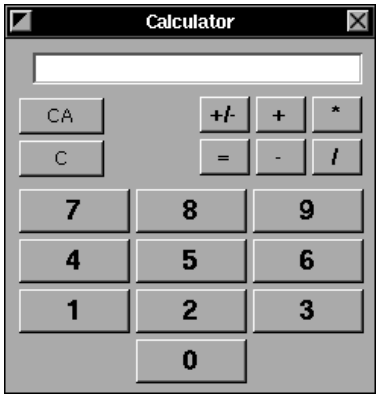
```
- enterOp:sender
{
   if (yFlag) {        /* something is stored in Y */
      switch (operation) {
         case PLUS:
            X = Y + X;
            break;

         case SUBTRACT:
            X = Y - X;
            break;

         case MULTIPLY:
            X = Y * X;
            break;

         case DIVIDE:
            X = Y / X;
            break;
      }
   }

   Y     = X;
```

```
    yFlag = YES;

    operation = [ [sender selectedCell] tag];
    enterFlag = 1;

    [self displayX];
    return self;
}
```

The **enterOp:** method is the real heart of our Calculator. It performs the arithmetic operation that was stored in the **operation** instance variable, sets up the registers and flags for another operation or another button click, and then displays the contents of the X register on the screen.

3. Activate Interface Builder and create a **Matrix** with two rows and three columns in the upper-right hand corner of your Calculator window. See Figure 20 below.
4. Set the title of each button to correspond with one of the six basic functions, as in Figure 20 below. Use a larger font (e.g., Helvetica Bold 16 pt.) so the titles are easily readable. Don't worry about the unary minus "**+/-**" button for now.
5. Set the tag of each button to correspond with the **enum** defined in the **Controller.h** file above.
6. Connect the function **Matrix** to the **Controller** by Control-dragging from the **Matrix** to the **Controller** instance icon and double-clicking the **enterOp:** action in the Matrix Inspector. This connection is similar to the one we made for the numeric keypad.

FIGURE 20. The Calculator Window with Six Function Buttons



7. Save the **Calculator.nib** file in IB by typing Command-s.

8. Compile your program and run it. If you use PB, all you have to do is click the **Run** button and PB will **make** the program when necessary. All buttons except the unary minus should work as you expect.

For easy access, we recommend that you keep the Project Builder, Interface Builder, Edit, Terminal, and Librarian, and HeaderViewer application icons in your dock while developing applications.

When you want to switch applications, use NeXTSTEP's **Hide** command rather than **Quit** to suspend the current application. This keeps your screen clear, and avoids the wait of having applications start up again when you need them. A better technique is to use NeXTSTEP 3.0's new "activate and hide all others" command. If you press a Command key and double-click a docked application icon, then the application activates (or launches) as usual and *all other active applications will hide themselves!*

Adding Unary Minus to the Controller Class

We want the unary minus function (the button with the "**+/-**" on it) to change the sign of the number currently displayed in our Calculator's **readout**. One way to implement this function is to handle it with another **case** in the **switch** statement in the **enterOp:** method. You could give the "**+/-**" key its own tag and have the **enterOp:** method intercept it and perform the appropriate function. The problem with this approach is that the unary minus function has virtually nothing in common with the other arithmetic functions: it takes one argument instead of two and operates immediately on the displayed value. A far better way for implementing this function is to implement a new action method in the **Controller** class.

Using IB's Parse Command with a New Action Method

Adding new action methods to existing classes is slightly more difficult than creating the initial class definition. You wouldn't want to use Interface Builder to create the new methods as we did before, because IB's **Unparse** command will *replace* the existing class files (**Controller.h** and **Controller.m** here) and wipe out the source code you've added. Instead, you should edit the class files to add the new methods that you want, and then command Interface Builder to **Parse** the files to learn about the changes.

1. Using an editor, insert the **doUnaryMinus:** action method definition in **`bold`** below into **Controller.h**. (You can tell it's an *action* method because **sender** is the only argument.)

```
...
- clear:sender;
- clearAll:sender;
- enterDigit:sender;
- enterOp:sender;
- displayX;
- doUnaryMinus:sender;
@end
```

2. Using an editor, insert the entire **doUnaryMinus:** method below into **Controller.m**.

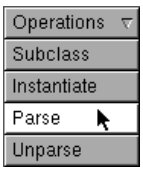
```
- doUnaryMinus:sender
{
   X = -X;
   [self displayX];
   return self;
}
```

It doesn't matter where you put this method in **Controller.m**, as long as it's between the directives **@implementation** and **@end**. However, we suggest you order the method implementations in the same way as the method declarations are ordered in the **Controller.h** class interface file.

Lastly, we have to tell Interface Builder about the new **doUnaryMinus:** method and set up a connection between the on-screen unary minus button and the **Controller**.

3. Activate IB and click the **Classes** suitcase icon in IB's File window to open the Classes view.

4. Select **Controller** in the class hierarchy in the Classes browser; recall it's a subclass of **Object** so you may have to scroll to the left.



5. Drag to **Parse** in the **Operations** pull-down list and you'll get the Parse panel in Figure 21 below. This panel tells us that the definition for **Controller.h** will be parsed from the edited file on disk.

6. Click **OK** in IB's Parse panel to parse the **Controller.h** file on disk. You should see the new **doUnaryMinus:** action method in the ButtonCell Inspector. See Figure 22 below.

7. Click the **Objects** suitcase icon in IB's File window to open the Objects view.

8. Double-click the unary minus button in the Calculator window. Make sure that the button, not the **Matrix** as a whole, has been selected.

FIGURE 21. Interface Builder's Parse Panel

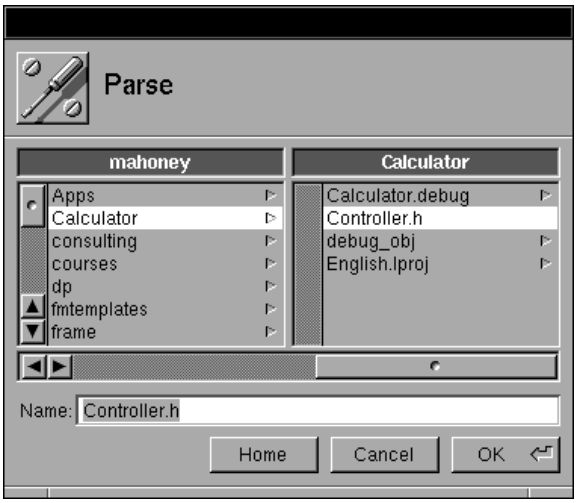
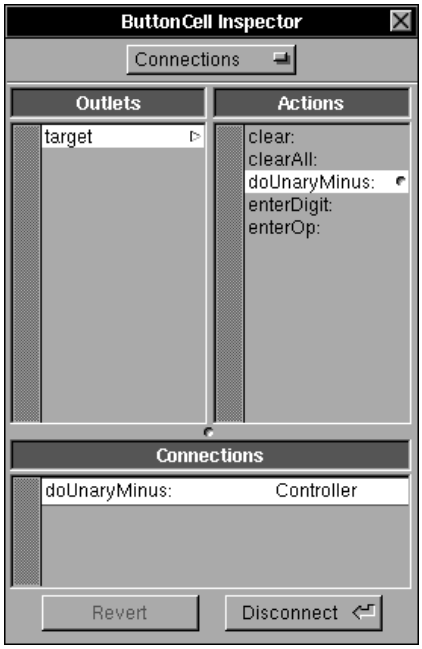


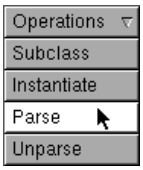
FIGURE 22. Parsed doUnaryMinus: Action Method in Inspector



9. Connect the unary minus button to the **Controller** by Control-dragging from the button to the **Controller** instance icon and double-clicking the **doUnaryMinus:** action in the ButtonCell Inspector. See Figure 22.

When a button in a **Matrix** object has its own target, the button's target overrides the target of the **Matrix**. So, when the user clicks on the unary minus button, the button will send the **doUnaryMinus:** message to its own target, rather than sending the **enterOp:** message to the target of the **Matrix**.

10. Save the **Calculator.nib** file (Command-s), **make** and run the program. The unary minus function should behave as expected.



You might be wondering why IB's **Parse** operation didn't bring in the definition of the **displayX** method in addition to the **doUnaryMinus:** method. The reason is that IB only looks for *action* methods when parsing a class interface (**.h**) file. An action method is a method declared in the form

```
- methodname:sender;
```

with a single argument called **sender**. As we'll see later, IB will also bring in *outlet* declarations when parsing a class interface file. These outlet declarations must be instance variables of the form

```
id outletname;
```

Action methods and outlets are the *only* types of information that Interface Builder learns about a class when it parses a class interface file!

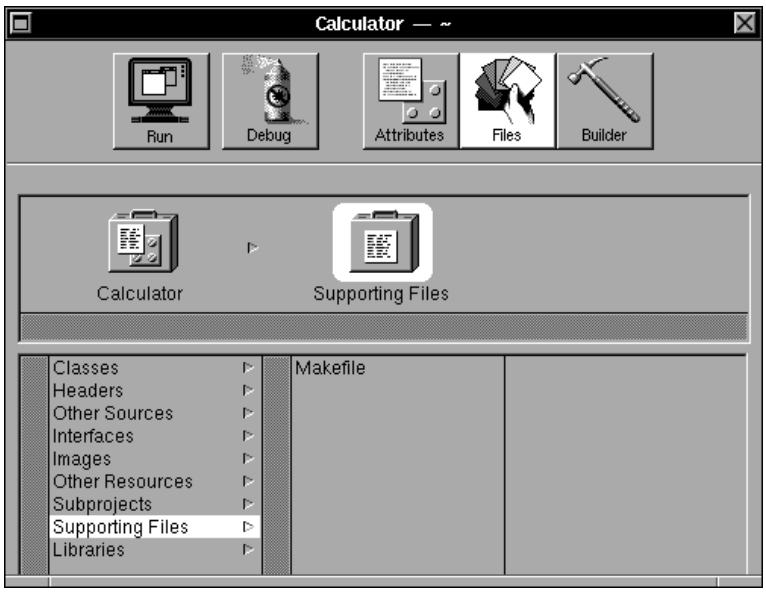
The Files in a Project

If you've been checking your **~/Calculator** directory while stepping through this chapter you've noticed that several files were automatically created in it. In this section we'll discuss what these files contain and how they fit into a project.

Project Builder's Files view uses a browser to list each project file by type, as in Figure 23 below. You can change which type of files are displayed by clicking a file type (e.g., **Images**) in the left-hand column of the browser. In the table below we briefly discuss what each file type means.[1]

1. In NeXTSTEP 2.1, Interface Builder's built-in Project Inspector uses a similar browser to list each project file type by its file extension.

FIGURE 23. Supporting Files in Project Builder



Several files are automatically placed in your project and project directory when you create a new project. Some of these, for example, **Makefile**, the **main()** function file (e.g., **Calculator_main.m**), and the Application Kit libraries, show up in PB's Files View.

File Type	Typical Extensions	Meaning
Classes	**.m**	Objective-C class implementation (e.g., **Controller.m**) source code files that you've created.
Headers	**.h**	Objective-C class interface (e.g., **Controller.h**) and C header files that you've created.
Other Sources	**.c, .m**	ANSI C (**.c**) and Objective-C (**.m**) source code files (e.g., **Calculator_main.m**).
Interfaces	**.nib**	NeXTSTEP Interface Builder files, which contain information on objects created in IB.

File Type	Typical Extensions	Meaning
Images	**.tiff, .eps**	TIFF (tag image file format, or bitmap) images and Encapsulated PostScript files.
Other Resources	**.psw, .snd**	Other resource files which don't fit any of the other types.
Subprojects	**.subproj**	Interface Builder subprojects.
Supporting Files		**Makefile** and any others.
Libraries	**.a**	Library files that are linked into your program (e.g., **/usr/lib/libNeXT_s.a**)

The Project Builder-generated Makefile

Every NeXTSTEP a *project* directory needs a special file called *Makefile* (we discussed Makefiles and **make** targets (e.g., **make debug**) briefly in Chapter 2). The Makefile contains the specifications needed to tell the standard UNIX utility called **make** how to create directories, call the **cc** compiler, and do other tasks associated with building an application. If you've used Makefiles and **make** before, you'll be glad to know they work essentially the same way in NeXTSTEP; the big difference is that the Makefile is generated automatically for you by Project Builder. That's good news, because writing a Makefile in the traditional UNIX environment can be tedious and time-consuming.

The Makefile contains the names of all of the **nib**, Objective-C, **tiff** (bitmap), **snd** (sound), **psw** (PostScript), library, and other files that get linked together to produce your final application program. It can also contain installation information, include file dependencies, and other information.

Let's take a close look at the Makefile that Project Builder generated for us when we chose **Project→New** to create the Calculator project.

```
# Generated by the NeXT Project Builder.
#
# NOTE: Do NOT change this file --
#    Project Builder maintains it.
#
# Put all of your customizations in files called
# Makefile.preamble and Makefile.postamble
# (both optional), and Makefile will include them.
```

```
NAME = Calculator

PROJECTVERSION = 1.1
LANGUAGE = English

LOCAL_RESOURCES = Calculator.nib

CLASSES = Controller.m
HFILES = Controller.h
MFILES = Calculator_main.m

OTHERSRCS = Makefile

MAKEFILEDIR = /NextDeveloper/Makefiles/app
INSTALLDIR = $(HOME)/Apps
INSTALLFLAGS = -c -s -m 755
SOURCEMODE = 444

ICONSECTIONS = -sectcreate __ICON app \
    /usr/lib/NextStep/Workspace.app/application.tiff

LIBS = -lMedia_s -lNeXT_s
DEBUG_LIBS = $(LIBS)
PROF_LIBS = $(LIBS)

-include Makefile.preamble
include $(MAKEFILEDIR)/app.make
-include Makefile.postamble
-include Makefile.dependencies
```

Here's what most of the lines in this Makefile mean:

NAME=	Tells **make** the name of your application.
LOCAL_RESOURCES=	Lists the names of all nib files in your project.
CLASSES=	The Objective-C classes you've added.
HFILES=	The Objective-C and C header files added.
MFILES=	The Objective-C files that are not class implementation files.
MAKEFILEDIR=	Specifies the directory, **/NextDeveloper/Makefiles/app**, that should be searched for any Makefiles that are included. The standard included NeXT-STEP Makefile is **app.make**.

INSTALLDIR=	Specifies the directory where your program should be copied when you type "**make install**."
INSTALLFLAGS=	Specifies the flags passed to the **install** program when you type "**make install**."
SOURCEMODE=	When source code is archived in a directory in response to "**make installsrc**," SOURCEMODE specifies the UNIX file protection mode that these files should have. Mode 444 is read-only for all users.
LIBS=	Specifies which libraries should be linked into your program. The standard libraries are **libNeXT_s.a** and **libMedia_s.a**; they specify the NeXTSTEP Application Kit and a UNIX library.
DEBUG_LIBS=	Specifies which libraries should be used when the application is built in "debug mode."
ICONSECTIONS=	This line specifies the names of icons that should be linked into your application. They'll end up as sections of the executable file. **application.tiff** refers to the generic application icon we saw in Figure 18 above.
-include Makefile.preamble	Causes **make** to include the file called **Makefile.preamble**, if it exists in your project directory.
include $(MAKEFILEDIR)/app.make	Includes the **/NextDeveloper/Makefiles/app/app.make** file, which is the standard NeXTSTEP Makefile for compiling NeXTSTEP applications.
-include Makefile.postamble	Causes **make** to include the file called **Makefile.postamble**, if it exists in your project directory.
-include Makefile.dependencies	Causes **make** to include the file called **Makefile.dependencies**, if it exists in your project directory. This file is created by typing "**make depend**."

Since Project Builder (IB in NeXTSTEP 2.1) updates your project's Makefile, and since the Makefile is used to compile your program, it's important to keep PB up-to-date. When you add a file to your project, do it by using the PB, rather than modifying the Makefile directly. Indeed, you should *never* edit the Makefile directly; if you need to add commands not available via the Project Inspector, then put them in one of the files **Makefile.preamble** or **Makefile.postamble**, which are included in the Makefile if they exist in the project directory.

You can use the file **Makefile.dependencies** to tell **make** about dependencies created by **#include** and **#import** statements in your source code. This will force **make** to recompile a file if any of its included files are modified. From a UNIX shell command line, type "**make depend**" to create the file **Makefile.dependencies**: this will cause the C preprocessor to search for **#include** and **#import** statements and create a dependency file that **make** will include automatically.

To learn more about Project Builder's use of Makefiles, look at the master NeXTSTEP Makefile in **/NextDeveloper/Makefiles/app/app.make**.

The PB-generated main() Program File

We've seen that Project Builder (IB in NeXTSTEP 2.1) will generate a Makefile according to our specifications. PB will also generate an Objective-C file, **Calculator_main.m** in our example, containing an program's **main()** function. The **main()** function is where every Objective-C (and C!) program begins. Below we list the generated **Calculator_main.m** file.

```
/* Generated by the NeXT Project Builder
   NOTE: Do NOT change this file --
   Project Builder maintains it.
*/

#import <appkit/Application.h>

void main(int argc, char *argv[]) {

  NXApp = [Application new];
  if ([NXApp loadNibSection:"Calculator.nib"
       owner:NXApp withNames:NO])
    [NXApp run];

  [NXApp free];
  exit(0);
}
```

Below we briefly discuss the executable statements in this **main()** function.

NXApp = [Application new];
This line contains a message to the **Application** class which creates a new **Application** object instance, and therefore sets up a connection to the Window Server. (Recall that every NeXTSTEP application needs precisely one **Application** object.) The **id** of this new object is assigned to the variable **NXApp**. **NXApp** is a global variable declared in **Application.h**, to make it easier to send messages to it from any part of your program.

[NXApp loadNibSection:"Calculator.nib"
owner:NXApp withNames:NO]
This line sends the **loadNibSection:owner:withNames:** message to **NXApp**, which causes the **Calculator.nib** file to get "loaded" from the **Calculator.app** (or **Calculator.debug**) directory into your application's memory map. The objects that were specified in Interface Builder will be created in memory and any initialization that you've defined for these objects (e.g., the size of a button, an outlet to another object) automatically gets executed at this time.

[NXApp run];
This line sends the **run** message to **NXApp**, which causes the application's (**Application** object's) main event loop to run. The main event loop handles menu clicks, keystrokes and all of the other events to which an application can respond. Control doesn't return to **main()** until the object **NXApp** gets a **stop:** or **terminate:** message, usually in response to a user choosing the application's **Quit** menu item.

[NXApp free];
This line sends the **free** message to **NXApp**, which closes all the application's windows, breaks the connection to the Window Server, and frees the **Application** object that was created when the **new** message was sent to the **Application** class.

exit(0);
The standard C **exit()** library function which gracefully ends the application.

Every **main()** program file generated by NeXTSTEP has these same lines, although the name of the nib loaded is usually different. (Sometimes the class name **Application** is replaced by a subclass of the **Application** class.)

Other PB-generated Files

In addition to creating the Makefile, nib, and **main()** program files for a project, Project Builder created the following files for us:

- **PB.project** – a project file to keep track of all the parts of the project. If you look at this ASCII file in an editor, you'll see a lot of the same information as a Makefile.[1]
- **Calculator.iconheader** – contains information about application and document icons in your project; we didn't specify any of these icons in this chapter, so it contains the default icon names.
- **PB.gdbinit** – a **gdb** initialization file for this project.
- **English.lproj** – a *directory* which contains the information for an English language version of our project, including the **Calculator.nib** file in our example.

Summary

We did quite a bit in this Chapter! We started by building a "real" project with Project Builder. Then we used Interface Builder and Objective-C to build user interface objects, create and customize our own class, and connect these user interface objects with an object of our new class. We also learned a little more about Objective-C and some AppKit classes, and a lot about the files that PB generates. In the process we used all four of the operations that IB can perform on classes: **Subclass, Instantiate**, **Unparse**, and **Parse**.

In the next chapter, we'll add an Info Panel and some icons to our Calculator application and find out how to increase the efficiency of a NeXTSTEP application by using separate nib files.

---

1. In NeXTSTEP 2.1, the project file is called **IB.proj**, and there is no **PB.gdbinit** file or **English.lproj** directory.