www.realbasic.com

Tutorial

REALbasic 2006 Tutorial

Documentation by David Brandt.
Concept by Geoff Perlman.

Printed in U.S.A.

Mailing Address	REAL Software, Inc. 1705 South Capital of Texas Highway Suite 310 Austin, TX 78746
Web Site	http://www.realsoftware.com
ftp Site	ftp://ftp.realsoftware.com
Support	Submit via REALbasic Feedback at www.realsoftware.com
Bugs/Feature Requests	Submit via REALbasic Feedback at www.realsoftware.com.
Sales	sales@realsoftware.com
Phone	512-328-REAL (7325)
Fax	512-328-7372

Version 2006r3, June, 2006.

Contents

CHAPTER 1

Introducing REALbasic

Welcome to REALbasic!

REALbasic is an integrated development environment based on a modern version of the BASIC programming language. REALbasic is made up of a rich set of *graphical user interface* objects (commonly referred to as GUI), an object-oriented language, a debugger, and a cross-platform compiler.

REALbasic provides you with all the tools you need to build virtually any application you can imagine.

If you are new to programming, you will find that REALbasic makes it fun and easy to create full-featured applications. If you are an intermediate or advanced programmer, you will appreciate REALbasic's rich set of built-in tools.

How to Use this Manual

The *REALbasic Tutorial* comprises a series of practical lessons for learning REALbasic. The lessons are structured so that they can be completed in an average of 30 minutes or less. Since the material in each chapter builds on the previous one, you should plan on working sequentially through this tutorial.

During the course of this tutorial, you will use REALbasic to build a complete application. You will build a text editor application that is similar to NotePad, the text editor that is included with Windows or SimpleText, the text editor that ships with Mac OS X. Using REALbasic, you will be able to compile the application for Windows, Macintosh, and Linux computers.

You will quickly learn to appreciate REALbasic's power and ease of use. For the entire application, you will only need to create about 200 lines of programming code.

Who Should Use this Manual

The REALbasic *QuickStart* and this *Tutorial* are written for someone who is new to programming. If you have never programmed before, check out the *QuickStart* before starting in on the *Tutorial*. You do not need any knowledge of programming in order to complete this tutorial.

If you have programming experience, you may want to quickly review this tutorial so that you'll become familiar with REALbasic's integrated development environment (IDE) and language features.

If you are new to computers, you should study the documentation that came with your computer. The documentation will help you learn how to use the mouse, menus, disks, and other aspects of your computer.

Presentation Conventions

The Tutorial uses screen snapshots taken from the Windows, Macintosh, and Linux versions of REALbasic. The interface design and feature set are identical on all platforms, so the differences between platforms are cosmetic and have to do with the differences between Windows XP's standard appearance setting, the Mac OS X interface, and the Linux "Bluecurve" desktop.

Italic type is used to emphasize the first time a new term is used, and to highlight important concepts. In addition, titles of books, such as *REALbasic User's Guide*, are italicized.

When you are instructed to choose an item from one of REALbasic's menus, you will see something like "choose File ▶ New Project". This is equivalent to the instruction "choose New Project from the File menu."

The items within the parentheses are *keyboard shortcuts* and consist of a sequence of keys that should be pressed in the order they are listed. On Windows and Linux, the Control key is the modifier; on Macintosh, the Command key is the modifier. For

example, the shortcut “Ctrl+O” means to hold down the Ctrl key, press the “O” key, and release the Ctrl key. “⌘-O” is the Macintosh keyboard equivalent. It means to hold down the Command key, press the “O” key, and then release the Command key.

This icon means that the text to the right of it is supplemental information that clarifies a point or is relevant only to some REALbasic users.

When you see a paragraph with an exclamation point to its left, you should pay careful attention to the paragraph contents. This style of paragraph is used to give you warning messages, or essential information.

A paragraph with an icon to its left like this lets you know that a series of instructional steps follows:

1. This is a sample step.
2. This is a second sample step in this set of instructions.
3. Hoping not to be left out, the third step is included with the other two steps.

Bold is used to indicate text that you will type while using REALbasic.

Some steps ask you to enter lines of code into the REALbasic Code Editor. They appear in Frutiger (a sans serif font) in a shaded box:

```
//update Font Size menu
If Str(Me.SelTextSize) <> SizeMenu.Caption then
  SizeMenu.Caption=Str(Me.SelTextSize)
  SizeMenu.MenuValue=SetFontSizeMenu(Me.SelTextSize)
End if
```

When you enter code, please observe these guidelines:

- Type each printed line on a separate line in the Code Editor. Don’t try to fit two or more printed lines into the same line or split a long line into two or more lines.
- Don’t add extra spaces where no spaces are indicated in the printed code.

Occasionally a logical line of code is too long to fit on one line in the printed manual. When this happens, the last character of the first line is an underscore and the text that is continued on the next line is indented. It appears like this:

```
MsgBox "The text you are searching for"_
                +chr(210)+Value+chr(211)+" could not be found."
```

REALbasic uses the underscore character to indicate that a logical line of code will be continued onto the next line in the editor. You can split up the logical line into two lines as long as you use the underscore character as shown in the examples. When you use the underscore character correctly, REALbasic will automatically indent the text on the following line, as shown above.

Whenever you run your application, REALbasic first checks your code for syntax errors as described in the section, "Automatic Debugging Features" on page 108. Syntax checking will direct your attention to the line of code that is causing problems. Check the line against the printed line. Also, if you have trouble getting your code to work, you can always open the lesson file for that chapter (described in the next section) and paste the corresponding code into your project.

Lesson Files

REALbasic files for each completed chapter are included on the CD in the "Tutorial Files" folder. You can compare your work at different stages of this tutorial with that given in the provided files. You can also start a new chapter using the completed file from the previous chapter.

Since completed REALbasic project files are provided for each chapter, you can skip over a chapter if you get stuck. Later, you can easily return to a particular chapter to revisit the material.

On Your Mark, Get Set, Go!

You are now ready to begin learning REALbasic!

CHAPTER 2

Creating Windows

In this chapter you will be introduced to REALbasic and its Integrated Development Environment (IDE). You will learn how to:

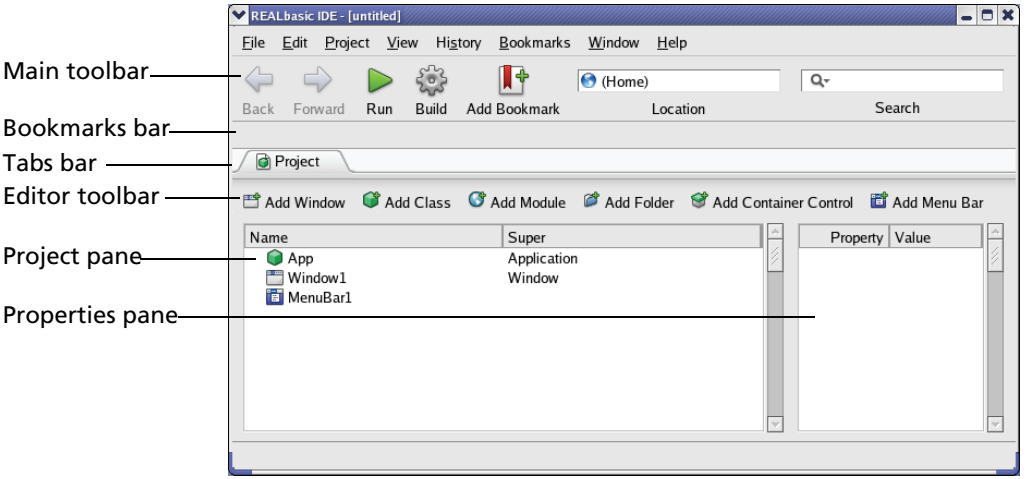
- Start Up REALbasic
- Identify REALbasic's windows
- Build a document window that will hold the text of your text editor
- Run your application

Starting Up REALbasic

Locate the REALbasic application icon on your computer (it's in the folder in which you installed REALbasic), and double-click it to start up REALbasic.

When you start REALbasic, the main IDE window appears. This is shown in Figure 1.

Figure 1. The REALbasic IDE window.



The IDE window resembles an internet browser window. The main work area of the window works like a internet browser that supports tabbed browsing, such as FireFox or Safari. When you first start REALbasic, only one tab exists; as you add items to your project, REALbasic automatically adds tabs to the IDE window.

The Main Toolbar contains controls for navigating among parts of your project. The Back and Forward buttons allow you to redisplay previously viewed screens, the Location area lets you go to an item by name, and the Search area lets you search for specific objects throughout your project.

With the Add Bookmark button, you can add frequently used items to the Bookmarks menu or the Bookmarks bar. You can navigate to bookmarked pages simply by choosing its name from the Bookmarks menu or clicking its name in the Bookmarks bar. You use the Run and Build buttons in the Main Toolbar to test and build your application.

The History menu tracks the series of screens that you have worked on. You can redisplay a screen by choosing its name from the History menu.

The body of the window displays parts of the project organized into screens. Each screen is labeled with a tab. When you first launch REALbasic, only one tab is available, the tab for the Project Editor, shown in Figure 1. Just below the tabs is a toolbar that belongs to the editor that is displayed in that screen. You use it to add or manipulate items in the editor you are viewing. The Editor toolbar changes depending on which screen you are on.

The Project Editor

When you start REALbasic, the Project Editor is shown in the IDE window. It contains two panes, the Project pane and the Properties pane. A moveable divider separates the two panes; you can resize the panes by dragging the divider to the left or right.

A *project* is the collection of items that make up the application you are developing. For example, the application's windows and menubars are project items. Some of the other items that might be listed in the Project Editor are classes, modules, pictures, sounds, databases, and movies. You use the Editor toolbar to add items to the project. You can also add items to the project by choosing an item from the Project ▶ Add submenu.

The left side of the Project Editor displays a list of project items. You can double-click on a project item to go to an editor for that item or, if there is no editor, a viewer.

The right side of the Project Editor contains the *Properties pane*. *Properties* are attributes of an item. For example, some of a window's properties are its size and position. The Properties pane contains the list of the names of properties and their values for the *currently selected* object in the Project pane. When you select a different object, the Properties Window changes to show the properties of that object. If no object is selected, the Properties pane is empty.

In Figure 1, no project items are selected so the Properties pane is blank. If you were to click on a project item, the Properties pane would change.

Building a Document Window

Now that the introductions between you and REALbasic are over, you can start building your own application!

When you start REALbasic, the Project Editor includes an item for the application's *default window*.The default window is the window that appears automatically when your finished application is launched. The text editor that you will build uses the default window as the main document window.

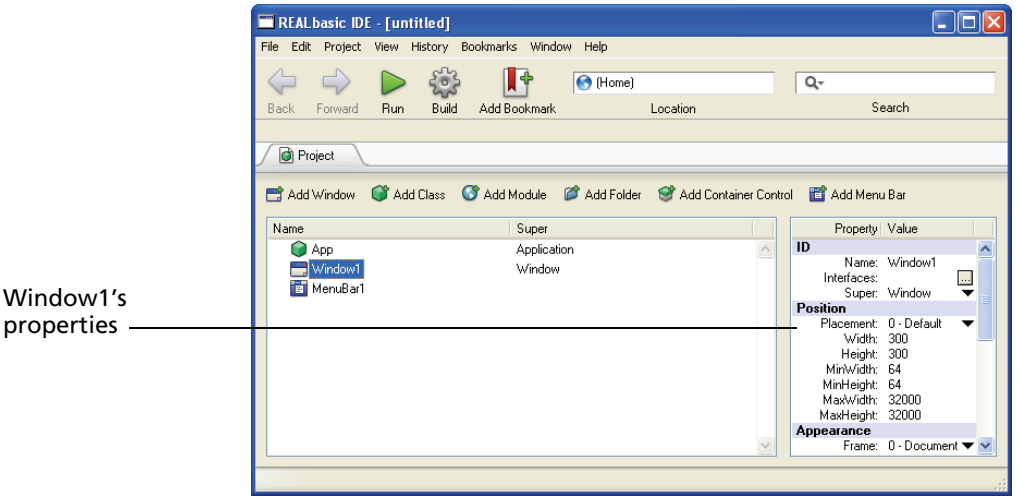
To design the main document window, you use IDE's *Window Editor*. With the Window Editor, you will need to place a text editing area in the window that is the same size as the window itself. You will do this in the following series of steps.

Since this will be the window that contains the text editor, we will first give it a more meaningful name.

To rename Window1, do this:

1 Click on Window1's name in the Project pane.
The Properties pane changes to show all of Window1's properties. It should look like Figure 2.

Figure 2. The Project Editor with Window1 selected.



In the "ID" group of properties, you will see the value of Window1's Name property.

2 In the Properties pane, change Window1's Name property to **TextWindow** and press the Enter key (Return on Macintosh).
Notice that its name changes in the Project pane.

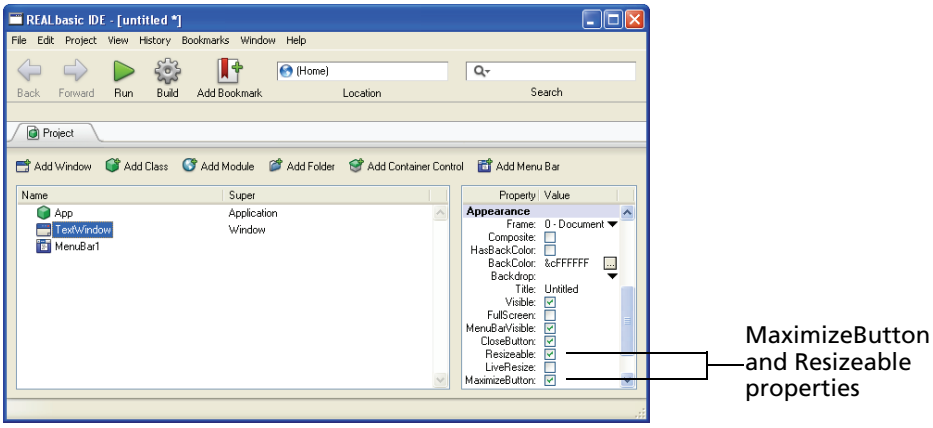
Later in the Tutorial, you will refer to TextWindow by name, so be sure to change it as shown here. Otherwise, REALbasic won't be able to find it.

Next, we need to tell REALbasic to add the standard Resize Box and Maximize button to TextWindow so the user can resize the window.

To configure TextWindow as a resizable window, do this:

- In TextWindow's Properties pane, check the Resizeable and MaximizeButton properties. They are located in the Appearance group.
The Appearance group should now look like Figure 3.

Figure 3. TextWindow's Appearance Properties Window.



The properties in the ID group in the Properties pane specify the Name of the window and the type of object it is. That is, the Super property tells us that it is a Window. You will want to assign meaningful names to the objects that you work with so that you can easily refer to them in your code.

Adding an EditField

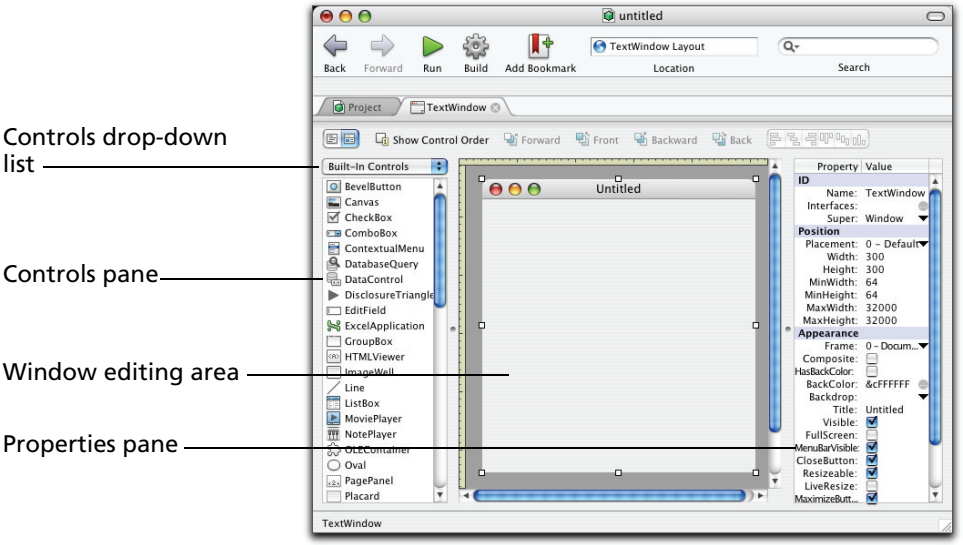
In order to make TextWindow capable of handling text, we'll use an interface object called an *EditField* control. This is the interface object that accepts text input from the end-user. The EditField control is one of the interface elements that are listed in the pane on the left side of the Window Editor.

To add an EditField to TextWindow, do this:

1 Double-click the TextWindow item in the Project Editor.

REALbasic opens the Window Editor for Window1. It also creates a new tab for Window1's Window Editor. Notice that there are now two tabs in the Tab bar and the Window Editor for Window1 is displayed.

Figure 4. The Window Editor for TextWindow.

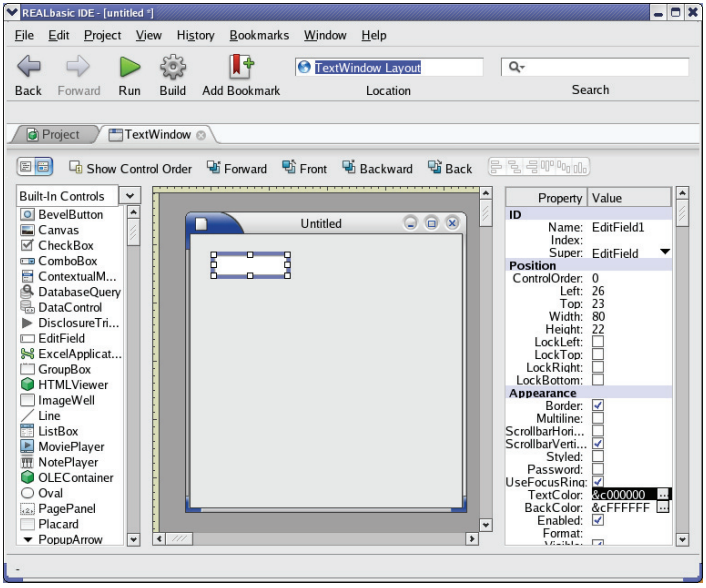


The list on the left side of the Window Editor contains the built-in controls that you can add to the window. Controls are interface elements such as pushbuttons, checkboxes, radio buttons, drop-down lists, lists (such as the controls list itself), as well as multimedia controls like the movieplayer and noteplayer.

The center area of the Window Editor is where you add interface elements to the window. The right pane is the Properties pane. It shows the properties of the selected item in the window being edited or the properties of the window itself. Since there are no items in Window1 yet, the Properties pane now shows TextWindow's properties. This is the same set of properties that you saw in the Project Editor.

2 Locate the EditField control in the Controls pane, click on it and drag it anywhere onto TextWindow.

Figure 5. The Window Editor after adding an EditField.



Since the EditField is selected, the Properties pane shows its properties

The EditField is REALbasic's text entry control. It can be configured to accept either a single line of text or multiple lines and paragraphs.

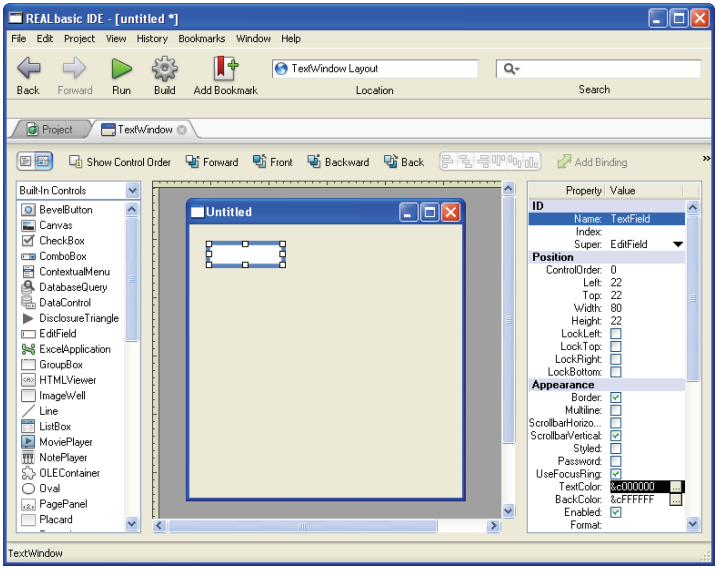
Since the EditField is the currently selected object, the Properties pane now shows its properties. If you deselect the EditField by clicking elsewhere in the window, the Properties pane changes to show TextWindow's properties.

You use the controls in the Controls pane as templates for the items in your windows. When you drag a control from the Controls pane to a window, REALbasic creates a clone based on the template. This clone automatically gets all the properties that belong to the template. This is what's shown in the Properties pane right now.

3 Use the Properties pane to change the Name property of the EditField from EditField1 to **TextField**.

The Window Editor should look like Figure 6.

Figure 6. The Window Editor after renaming EditField1.

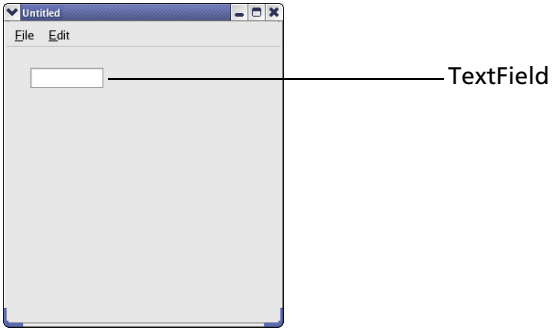


Although you have just started building your application, you may want to run it now, just to see what happens.

To run your application, do this:

1 Click the Run button in the Main Toolbar.
REALbasic compiles your application and launches it in its own window. The Text Editor application appears and should look similar to that shown in Figure 7.

Figure 7. The first run of your application.



2 Type something in the TextField to try it out.
Notice that the Edit menu's Cut, Copy, Paste, Delete (Clear on Macintosh), and Select All menu items work automatically.

3 After you are done exploring, choose File ▶ Exit on Windows and Linux or My Application.debug (Mac OS X) ▶ Quit on Mac OS X. On Windows and Linux, you can also quit by clicking the window's Close box to return to the REALbasic IDE.

When you click Run to launch your application, REALbasic compiles the code you've written. launches the application, and adds a new panel to the IDE for debugging that's called "Run." This is where you can monitor the execution of your code. Since there's no code yet, there is nothing of any interest in this screen. You can also quit out of the Debugging environment by closing the Run tab by clicking its close box.

In order to resume work within REALbasic's IDE, you must quit out of the test application.

Configuring TextField as a Text Editor

The TextField that you just created is, obviously, not an adequate text editor. In a text editor, the user can type as much text as he wishes. Also, the text editing area must be the same size as its window. Right now, the TextField you placed inside of TextWindow can handle only a small amount of text, all of which is on one line. To make a usable text editor, the TextField must have a scrollbar and accept many lines of text. In this section, you configure TextField so that it functions as a text editor and resize it so that its size matches its window.

In the first series of steps, you will use the Properties pane to move the TextField to the top-left corner of TextWindow.

To resize TextField so that it will handle multiple lines of text, do this:

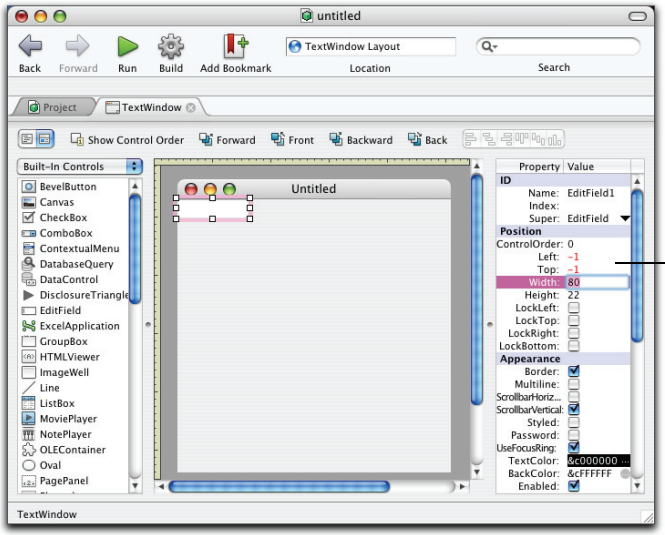
1 Click the EditField, TextField, in the Window Editor to select it, if it isn't already selected.

Refer to the Properties pane and locate the Top and Left properties. Delete the value corresponding to the Left property and replace it with **-1**. Press the Tab key to set the property value and select the next property, Top.

Notice how TextField moves to the left side of TextWindow. The value of -1 places the left edge of the TextField just outside the border of TextWindow.
The Properties pane shows negative values in red, making it easy to spot them.

2 Repeat step 2 for the Top property. Change its value to **-1**.
The TextField is now aligned with the top of TextWindow and should look like that shown in Figure 8.

Figure 8. The Window Editor after moving TextField.



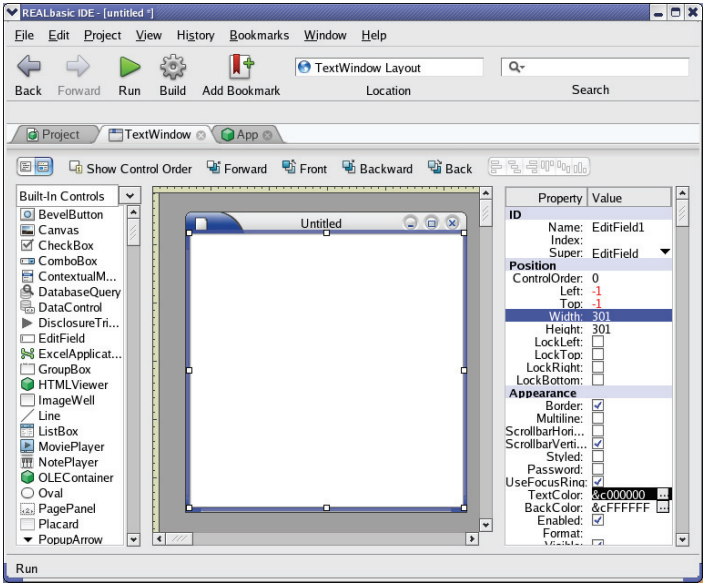
The Left and Top properties are in red, indicating negative values

3 Drag the resizing handle in the lower right corner of TextField to the bottom-right corner of the window, until it is close to the resizing handle of TextWindow.

4 To align the right side of the TextField with the right side of TextWindow, drag the resizing handle of TextWindow.

It should snap into place. The window should now look as shown in Figure 9:

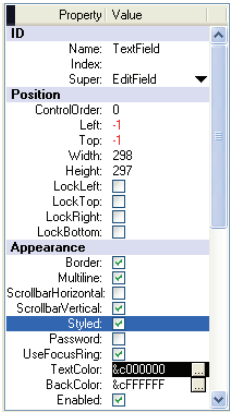
Figure 9. The TextWindow after resizing.



Next, you must tell REALbasic that you want TextField to accept as many lines of text as the user enters, display a scrollbar, and wrap text whenever a line of text reaches the right side of the TextField. This is also a good place to tell REALbasic that the EditField will display text of multiple fonts, font sizes, styles, and colors. This is done simply by selecting the MultiLine and Styled properties of the TextField. MultiLine enables several lines of text and Styled enables a mixture of fonts, font sizes, styles, and colors.

5 Select the MultiLine and Styled properties of the TextField by clicking their checkboxes in the Appearance group in the Properties pane.
The Properties pane should now look like this.

Figure 10. TextField's Properties pane after setting the MultiLine and Styled properties.



The TextField will now get a vertical scrollbar when the user enters more lines of text than can be displayed. When you add the ability to set font style attributes, TextField will now be able to display them.

To run the application again, do this:

1 Click the Run button in the Main Toolbar.
The text editing window appears.

2 Enter several lines of text.
As you type you will notice that your lines wrap as they reach the end of the line. After you type enough lines to fill the window, the scrollbar will become active. You can use the scrollbar to get back to the top. You don't yet have the tools to demonstrate the effects of the Styled property; that will come in Chapter 6.

3 When you're done typing, choose File ▶ Exit on Windows and Linux or My Application (Mac OS X) ▶ Quit to return to the REALbasic IDE. Or close the window on Windows or Linux.

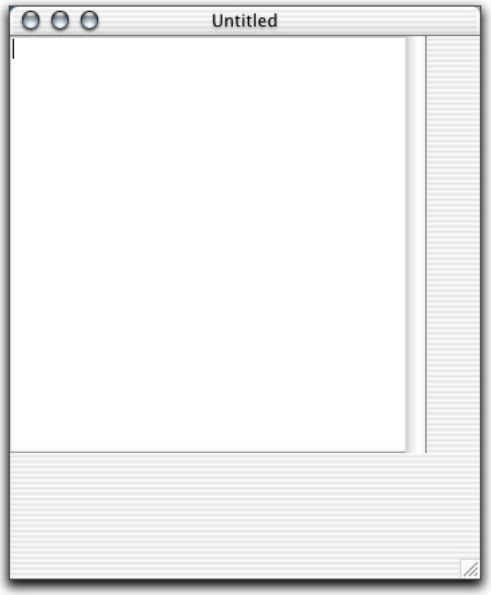
4 Save your project now. Choose File ▶ Save (Ctrl+S or ⌘-S) and give the project the name **TextEditor-ch2**.

NOTE: In the event that the computer crashes while you are testing your application, REALbasic will restore your project to its current state when you reopen the project. You don't need to save changes constantly to guard against lost work.

Lastly, you must configure TextField so that it remains the same size as its window when the user resizes the window using the window's Grow box. Unless you do this, the TextField and the Window will be the same size only if the user never resizes the window.

You can test this out by switching to the Debugging environment and resizing the window. You'll see something like Figure 11. Then choose Edit ▶ Undo to restore the window to its original state.

Figure 11. Resizing the document window with a fixed-sized EditField (Debugging environment).



In Figure 11, the EditField remains the size you specified in its Properties pane, but the window was resized by the user. Any user expects the editing area to be the same size as the window.

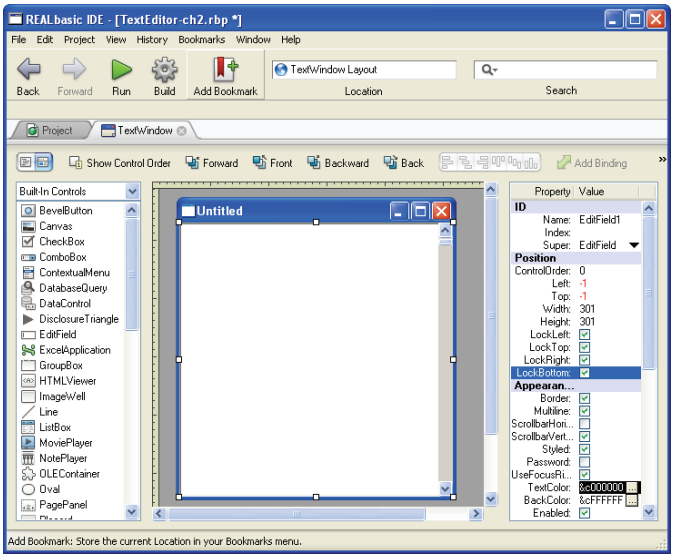
REALbasic provides a very simple way to accomplish this. The Lock properties are used to fix the distance between the edge of the window and the edge of the control. The distance between edges is maintained during resizing if the corresponding Lock property is selected.

To lock the size of the TextField to its window, do this:

1 Click on the TextField in the Window Editor. Locate the LockLeft, LockTop, LockRight, and LockBottom properties in the Position group in the Properties pane and select them using their checkboxes.

The Window Editor should now look as shown in Figure 12:

Figure 12. TextWindow's Window Editor after setting the Lock... properties.



2 Click the Run button in the Main Toolbar to test the resizing feature.

3 Quit out of the Debugging environment to return to the IDE.
On Windows and Linux, you can click the window's Close button to return to the REALbasic IDE.

4 In the IDE, choose File ▶ Save to save the settings of the four Lock... properties.

TextWindow includes a TextField—an object derived from the EditField control that is configured for text editing. TextField gets all the properties and methods of the EditField class automatically.

Any TextField is configured to accept multiple lines of text, display multiple styles of text, has a vertical scrollbar, and is locked to its parent window. When you create another instance of TextWindow, you get all the properties of TextField automatically. You're going to do this later on in Chapter 4 when you create a New item in the File menu.

Review

In this chapter you learned how to start up REALbasic, identify the windows of the Development environment, add a multiline EditField to a document window, lock it to its window, and run your application.

To Learn More About:	Go to:
REALbasic IDE	REALbasic User's Guide
REALbasic commands and language	REALbasic Language Reference

CHAPTER 3

Creating Menus

In this chapter you will work with menus in REALbasic. You will learn how to:

- Add a menu item to your application,
- Activate a menu item.

You can continue working from the application you began in Chapter 2 or open the application "TextEditor-ch2" in the Tutorial Files folder on the REALbasic CD.

Getting Started

If the TextEditor project is not already open, locate the REALbasic project file that you saved at the end of the last chapter ("TextEditor-ch2"). Launch REALbasic and open the project file. If you need to, you can use the "TextEditor-ch2" file that is in the Tutorial Files folder on the REALbasic CD.

Working with Text Documents

A text editor must be able to create, open, and save text documents. You will first add the ability to create new text documents. Implementing this menu item involves two steps:

- Adding the menu item to a menu,
- Adding a menu handler.

The first step adds the new item to the menu but the menu doesn't do anything until you add the menu handler. The menu handler is the set of instructions that is executed when the user selects the menu item. Every menu item that does something has to have a menu handler.

In this chapter, we will add a "New" menu item to the File menu.

Creating the New Menu Item

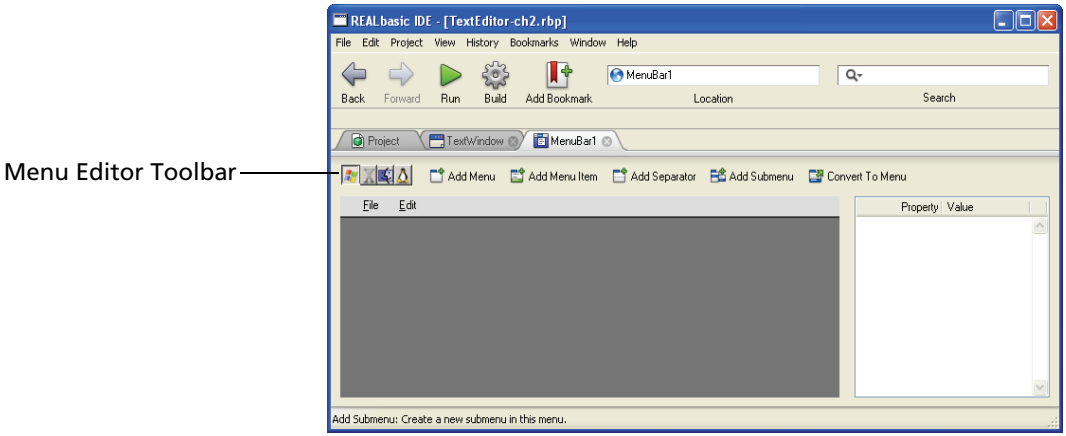
To create the New menu item, do this:

1. Click the Project tab in the IDE window to display the Project Editor.
2. Double-click the MenuBar1 item in the Project Editor to open its Menu Editor.

When you double-click the MenuBar1 item, REALbasic creates a new tab for the Menu Editor and displays it. You now have three tabs in the IDE, the Project Editor, the Window Editor for TextWindow, and the Menu Editor for MenuBar1.

MenuBar1 is the default menu bar that automatically applies to the whole application. If you wish, you can add other menubars and associate them with individual windows.

Figure 13. The Menu Editor.



The Menu Editor has its own Editor Toolbar with items that you use to add menus and menu items. The View Mode icons on the left side of the Editor Toolbar enable you to preview the menu system on the four platforms that REALbasic supports.

By default, the Menu Editor shows a preview of the menubar for the platform on which the REALbasic IDE is running. If you are running Windows or Linux, it shows the File and Edit menus. If you are running Mac OS X, it will also show the Apple and Application menus. You can preview the menubar on a different operating system by clicking the View Mode icon for another operating system. Your choices are Windows, Mac OS X, Mac OS "classic," and Linux.

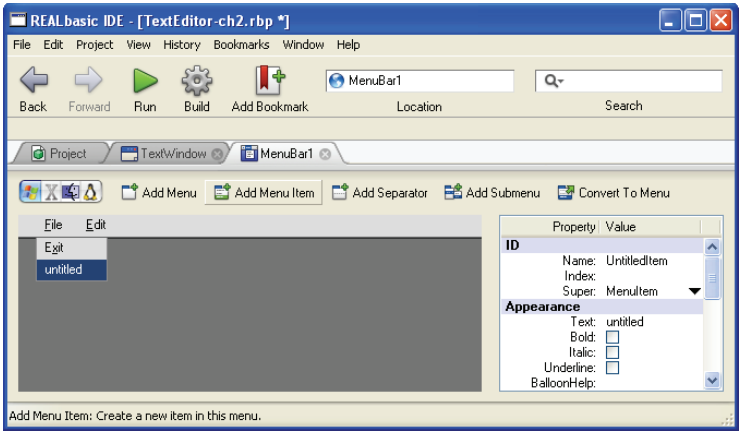
Figure 14. The View Mode icons.

3 Click on the File menu to display the default File menu items. By default, the File menu has only one menu item, for quitting the application. On Mac OS X, the Quit menu item will move to the Application's menu in the built application, but it is shown in the File menu within the Menu Editor.

4 With the File menu selected, click the Add Menu Item button in the Menu Editor Toolbar.

REALbasic adds a new menu item to the File menu and names it "untitled." The Menu Editor's Properties pane shows the properties of the new menu item.

Figure 15. The "Untitled" menu item in the Menu Editor.



5 Use the Properties pane to change the Text property of the menu item from "untitled" to **&New**. The ampersand prior to the letter "N" indicates that the letter "N" will be used as the keyboard accelerator on Windows and Linux. In the menu, the keyboard accelerator is underlined (Since keyboard accelerators are not used on Macintosh, the underline does not appear in the Mac OS X preview of the menubar).

Press the Enter key (Return on Macintosh) to save the entry.

Next, you will set the keyboard equivalent of the New menu item to Ctrl+N on Windows and Linux and ⌘-N on Macintosh.

6 In the Shortcut group, enter **N** as the Key property.

7 Check the MenuModifier checkbox if it is not already selected.

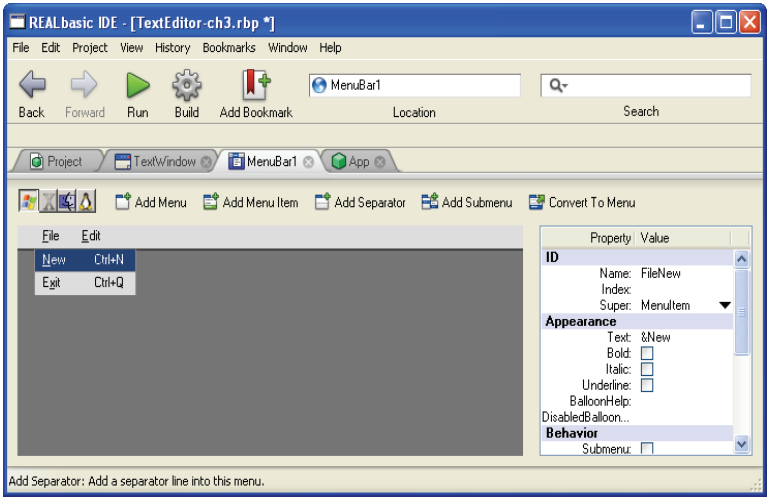
This means that the user must hold down the MenuModifier key when pressing the N to trigger the menu item. On Windows and Linux, the MenuModifier key is Ctrl; on Macintosh it is the Command key. If the MenuModifier key property was not selected, any press of the N key would trigger the menu command.

Lastly, we need to move the New menu item to its conventional position, at the top of the File menu.

8 In the Menu Editor, drag the New menu item above the Exit menu item.

Your Menu Editor should look like Figure 16. When you select the New menu item, the Properties window for that item should look as shown in Figure 16.

Figure 16. The Updated File menu.



Handling the New Menu Item

When you add a menu item, you need to tell REALbasic what to do when a user chooses the menu item. Right now, the user can select the New menu item, but absolutely nothing will happen.

You use the REALbasic language to tell REALbasic what to do when the user selects New from the File menu. The code that you write is called a *menu handler*.

The first decision you have to make is when the menu item should be available. Some menu items should be available all the time, while others should be disabled under certain circumstances. For example, if you want to add a Find menu item that displays a dialog box for searching for a string in the text editor document, it should be available and enabled only when the user has a document window open. If it were enabled when there is no document to search, it would not make any sense.

The New menu item should be available whenever the application is running. That is, the user should be able to create a new text document even if no document windows are open. For that reason, it has to belong to an object in the application

that is always available. It would be a mistake to associate the menu handler for the New menu item with the document window, TextWindow. That would mean that the menu handler would be available only when a document window is already open. This is a very important distinction on both Macintosh operating systems because the menu bar on a Macintosh is global to the whole application and is available when there are no application windows open.

REALbasic provides a place for code that must be available at all times, regardless of which windows are open—or even if no windows are open.

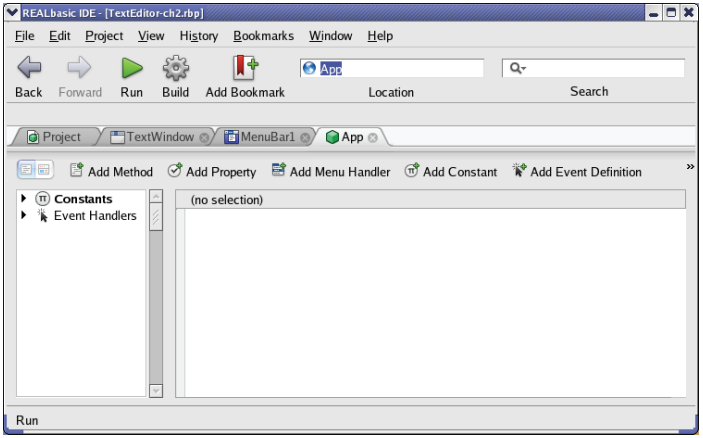
That place is the Application object. It is represented in the Project Editor as the "App" object.

REALbasic code that is placed in the Code Editor for the App object is available to the application as a whole—not just a particular window. To write the Menu Handler for the New menu item, first open the Code Editor for the App object.

To handle the New menu item, do this:

1. Click on the Project tab in the Tab bar to display the Project Editor.
2. Double-click the App object in the Project Editor.
REALbasic adds a new tab to the IDE window named "App". It contains the Code Editor for the App object. This is shown in Figure 17.

Figure 17. The Code Editor for the App class.



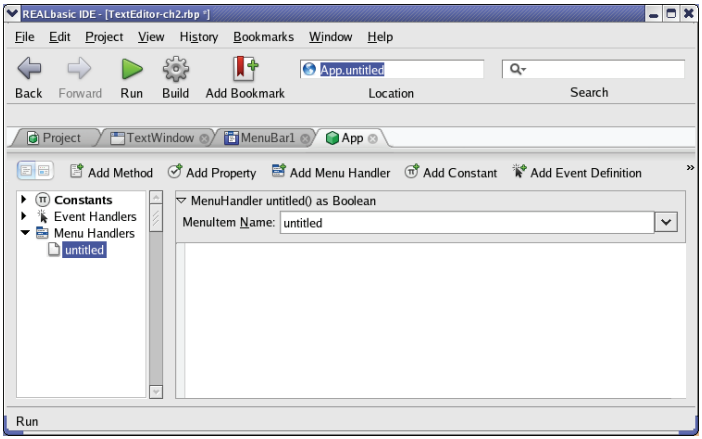
The Code Editor is divided into two panes. The pane on the left is a browser that enables you to select different items to edit. The larger pane is the body of the Code Editor; it shows the code associated with the item that has been selected in the browser. Since there is no code in this application yet, no code is showing and no items in the browser are selected.

We need to add a particular type of event handler called a menu handler. You use one menu handler per menu item. The first thing we need to do is tell REALbasic which menu item that we want to handle.

3 Click the Add Menu Handler button in the Code Editor toolbar.

REALbasic creates a Menu Handlers category in the browser and a new menu handler named "Untitled". When this menu handler is selected, the body of the Code Editor changes so that you can name the menu handler and enter its code. This is shown here.

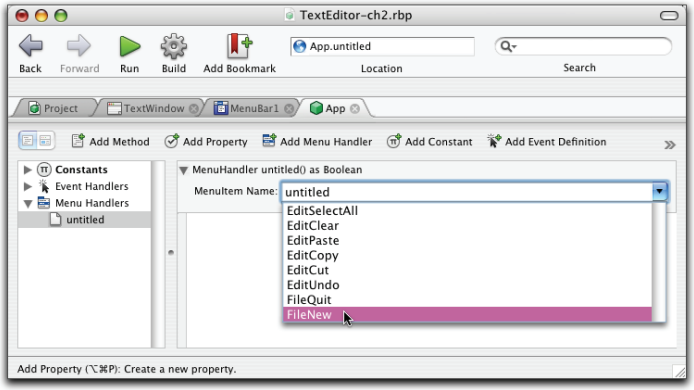
Figure 18. The new menu handler, "Untitled".



First, you need to choose the menu item that this menu handler will handle. The MenuItem Name drop-down list is already populated with the names of all the menu items in MenuBar1.

4 Use the MenuItem Name drop-down list to choose the FileNew menu item and press Tab to make the selection.

Figure 19. Choosing FileNew from the drop-down menu.



The name of the menu handler changes to FileNew in the browser area. This name must match the name of the menu item, as shown in Figure 16 on page 26.

5 In the body of the Code Editor, enter the following code for the menu handler:

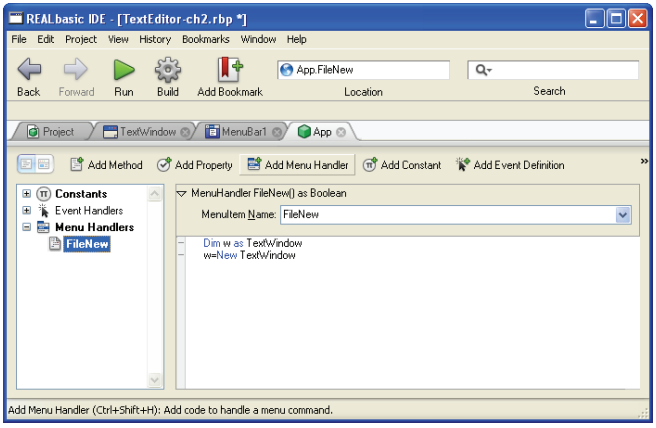
```
Dim w as TextWindow
w=New TextWindow
```

The Dim statement creates a new variable called "w". The term "Dim" is short for "Dimension" and it defines a new variable and declares the type of information that it will hold. In this case it is declared as a "TextWindow." A TextWindow is a type of window that we defined via setting its properties in the Properties pane and the items we added to the window in its Window Editor.

The next line creates a clone or an instance of the TextWindow, w. The New command in the second line creates the clone and returns it in the variable "w". The new copy, "w", is displayed immediately. This is because the Visible property of TextWindow is set to True in its Properties pane. True is the default value.

Your Code Editor should now look like this.

Figure 20. The menu handler for the New menu item.



6 Save your project as **TextEditor-ch3** and then click the Run button in the Main Toolbar to test the New menu item.

NOTE: If the application doesn't compile correctly, be sure you have renamed Window1 to TextWindow. If there is no item called "TextWindow" in the project, then REALbasic would not know how to handle the first line of code.

As you can see, the New menu item creates a clone of the original default text window with the properties you specified earlier in the tutorial. On Macintosh, try it out even if no text window is open.

7 When you are finished, quit the text application to return to the REALbasic IDE. On Windows and Linux, you need to close all of the windows you have opened to quit out of the TextEditor application. On Macintosh, you can chose the Quit menu item to quit no matter how many windows are open.

Review

In this chapter you learned how to add menu items to your application and how to write a menu handler.

To Learn More About:	Go to:
REALbasic Menus	REALbasic User's Guide: Chapters 3, 7.
REALbasic commands and language	REALbasic Language Reference

CHAPTER 4

Opening and Saving Documents

In this chapter you will work with documents in REALbasic. You will learn how to:

- Create menu items for opening and saving documents,
- Add code to your application to implement the menu items.

Getting Started

If the TextEditor project is not already open, locate the REALbasic project file that you saved at the end of the last chapter ("TextEditor-ch3"). Launch REALbasic and open the project file. If you need to, you can use the "TextEditor-ch3" file that is in the Tutorial Files folder on the REALbasic CD.

Saving Documents

In this section, you will add a Save menu item to the File menu. The procedure for adding the menu item to the project is the same as adding the New menu item, but the menu handler for the Save menu item does not belong in the App class. Since saving a document can occur only when a document is already open, the menu handler for the Save menu item should be added to the TextWindow object. This means that you will use TextWindow's Code Editor rather than the App class's Code Editor.

There is an additional issue that you will need to deal with. Even when a document window is open, the Save menu item should be enabled only when the document is

new and has never been saved or when there have been changes since the previous Save. Fortunately, REALbasic automatically monitors changes to the contents of TextWindow, so you will enable and disable the Save menu depending on REALbasic's analysis.

In summary, adding the Save menu item involves these operations:

- **Enabling and disabling the menu item.** The Save menu item should be enabled only when the contents of the current window have been changed—not all the time, as is the case for the New menu item.
- **Creating a menu handler.** The menu handler tells the TextWindow what to do when the user chooses the Save menu item. The menu handler that you will add calls a generic save-file method that actually accomplishes the save.
- **Adding a save-file method.** The save-file method is called by the menu handler. It uses a *FolderItem* object to manage saving the contents of the window to a text file on disk.

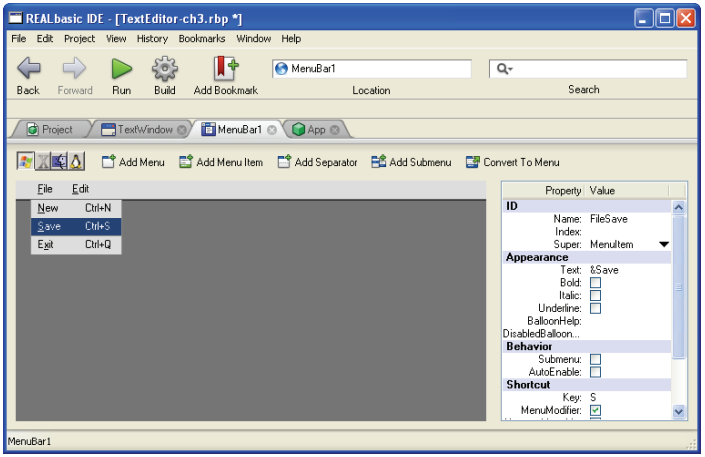
Adding the Save Menu Item

To add the Save menu item, do this:

1. Click the MenuBar1 tab to display the Menu Editor for MenuBar1, or, if the MenuBar1 tab is not open, click the Project tab and then double-click the MenuBar1 item in the Project Editor.
2. Click on the File menu to display its menu items.
3. Click the Add Menu Item button in the Menu Editor toolbar.
   REALbasic appends a new menu item to the File menu and names it "untitled."
4. In its Properties pane, change its Text property from "untitled" to **&Save**.
   The ampersand sets the Windows keyboard equivalent to "S".
5. In its Properties pane, deselect the AutoEnable property in the Behavior category.
   The AutoEnable property enables the menu item all the time; we used this for the New menu item, but this is not appropriate for a Save menu.
6. In the Properties pane, enter **S** as its Key property and click the MenuModifier checkbox.
7. In the Menu Editor, drag the Save menu item vertically between the New and Exit menu items (Windows and Linux) or the New and Quit menu items on Macintosh.

   The Menu Editor should now look like Figure 21 on page 33.

Figure 21. The updated Menu Editor.



Adding Properties to TextWindow

Before we write the menu handler for the Save menu item, we will add some properties to TextWindow that the menu handler will need.

As you know, TextWindow comes with a built-in set of properties because it based on the REALbasic object of type of Window. All windows have a basic set of generic properties. In addition, you can add properties of your own to TextWindow. A *property* of a window is a variable that can hold a value that describes some attribute of the window. Properties that you add to a window can store specific pieces of information that the window.

You will often want to add some specific properties to your windows that help you manage the window in your specific application. They are available to all the controls in the window.

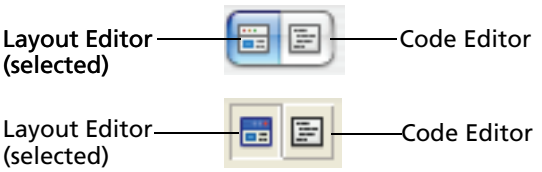
When we open a file in our application, we will need to keep track of the filename so that we can save changes. We will define a new property for TextWindow and store the filename in the property.

In this case, we need a variable that will store the filename of the document displayed in the window and a variable that will indicate whether the contents of the document have changed since the last Save to disk. Window properties can be assigned a value or can get the current value from any event handler in the window.

To add the properties to TextWindow, do this:

1. Click on TextWindow's Tab to display its Window Editor.
2. Click the Code Editor icon in the Editor Toolbar to display the Code Editor rather than the Window Editor.

Figure 22. The Edit Mode buttons for the Window Editor.



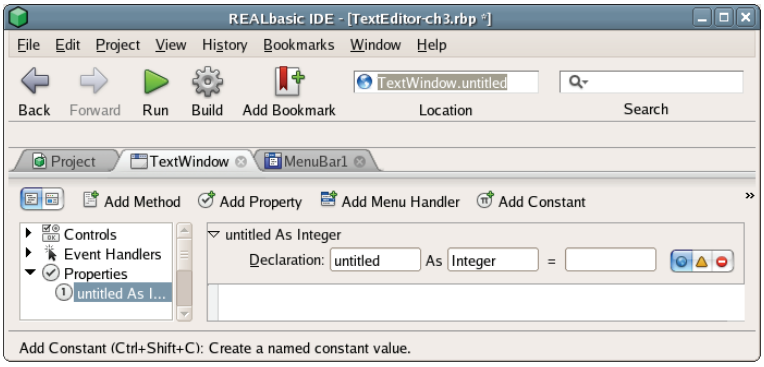
On Macintosh and Linux, the selected mode is highlighted, while on Windows the selected mode is depressed.

Since you are adding the properties to TextWindow—not App—its Code Editor needs to be active.

3 Click the Add Property button in the Code Editor Toolbar.

The Property Declaration area appears above the editing area. The term "declaration" means that you define the property by giving it a name and telling REALbasic what type of data it will store.

Figure 23. The Property Declaration area.



The Property Declaration area has three fields. They are for the Name of the property, its data type, and its initial value. The first two fields are required; REALbasic must know the name of the property (so that you will be able to refer to it in your code) and it also must know its data type.

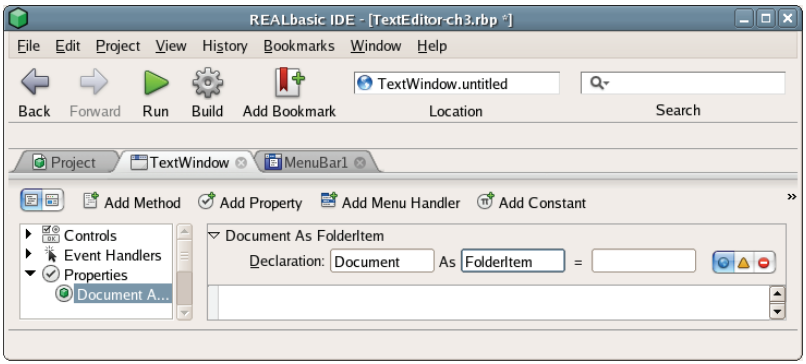
REALbasic is a "strongly" typed language, meaning that it checks to see that the values you try to store in variables and properties match the declared data type. You must declare the data type of every property and variable that you use; REALbasic will not attempt to figure out its data type from its usage. If you try to store a value that does not match the declared data type, you will get an error message when REALbasic attempts to compile the application.

When the Property Declaration area first appears, there is a default declaration for a property of type Integer. You can declare the new property by replacing (or accepting) these default values.

The field for the initial value is optional. If you leave this field blank, REALbasic will use the default value for the data type you declare.

4 Enter **Document** as the name of the property and **FolderItem** as the data type.
"Document" is the name of the property and "FolderItem" is its data type. In REALbasic, FolderItems are references to external files and directories (a.k.a., folders).

Figure 24. The Declaration for the Document propertuy.



5 Click the Add Property button in the Code Editor toolbar once again.
A new Property Declaration area appears above the code editing area.

6 Enter **TextHasChanged** as the property name and **Boolean** as the data type.
Boolean is a data type that can take only two values: *True* or *False*. In the section "Managing the TextHasChanged Property" on page 42 you will use this property to keep track of changes to the contents of the TextField. Whenever this property becomes True, REALbasic will know that it needs to enable the Save menu item.

The Document and TextHasChanged properties are now listed in the Properties group in the Browser area of TextWindow's Code Editor. They do not yet have assigned values. The Document property does not yet refer to an external file. Technically, its value is "Nil", meaning no value. TextHasChanged has its default value of False. Their values will be set via code.

Enabling the Menu Item

Since we want the Save menu item to be enabled only if there are unsaved changes to the document, the code will use the TextHasChanged property that you just added. The TextHasChanged property will let REALbasic know when the user has changed the contents of the TextField in TextWindow.

When you don't use the AutoEnable property of a menu item, it is disabled by default. This means you need to write some code to enable it when it is supposed to be enabled. You place this code in an EnableMenuItems event handler.

You can enable the menu item in either the App Code Editor or the TextWindow code editor. Since the Save menu should be enabled only when there is a document window open, we need to use the EnableMenuItems event handler for TextWindow.

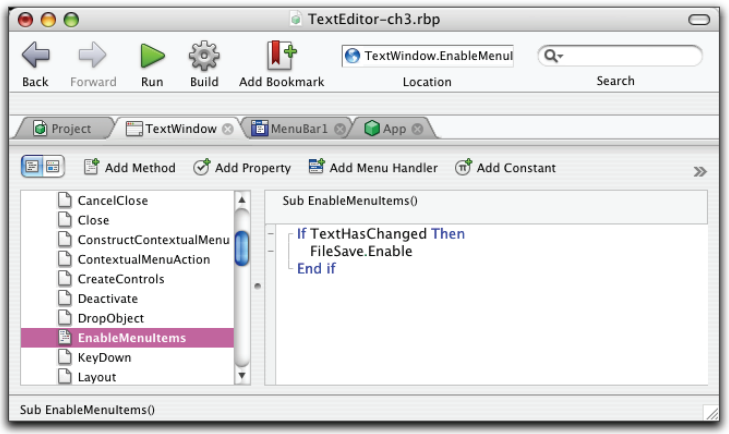
To enable the menu item, do this:

1. Click the plus sign (on Windows) or disclosure triangle (on Linux and Macintosh) next to the Event Handlers category label in TextWindow's Code Editor to reveal the events (or double-click the Event Handlers label).
2. Select the EnableMenuItems event and enter the following code into its Code Editor:

```
If TextHasChanged then
  FileSave.Enable
End if
```

The Code Editor should look as shown in Figure 25.

Figure 25. The EnableMenuItems Event Handler.



Notice that the code must be in TextWindow's Code Editor, not the App object's.

3. Save your project as **TextEditor-ch4**.

File Types

In order to manage opening and saving files, REALbasic needs to know about the types of files that the application can process. Obviously, you don't want a text editor application to try to open files of a type that it cannot display and users cannot edit.

The TextEditor application must be able to open, modify, and save the files it creates. On Windows it uses the RTF file type (RTF stands for Rich Text Format) and On Macintosh and Linux, it uses the TEXT file type. The Open-file dialog box should not display other file types, so the user will not be able to open files that the program cannot accept.

In order for your application to recognize specific types of files, you define the valid file types for the application. For example, if you are writing a graphics application, you will need to tell REALbasic that it needs to open files of type BMP, JPEG,

PICT, TIFF, and so forth. Also, if your application saves documents in its own format, REALbasic must know that file type as well.

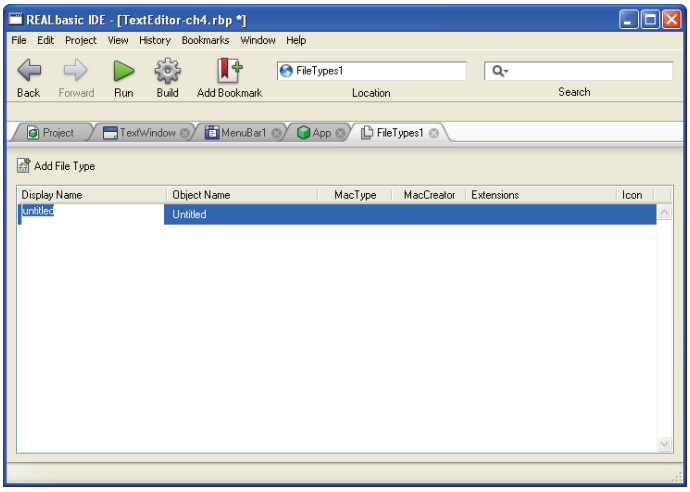
You use the File Type Sets Editor to specify valid file types. The following procedure adds this capability to the application.

Adding a File Type

To add the Text and RTF file types, do this:

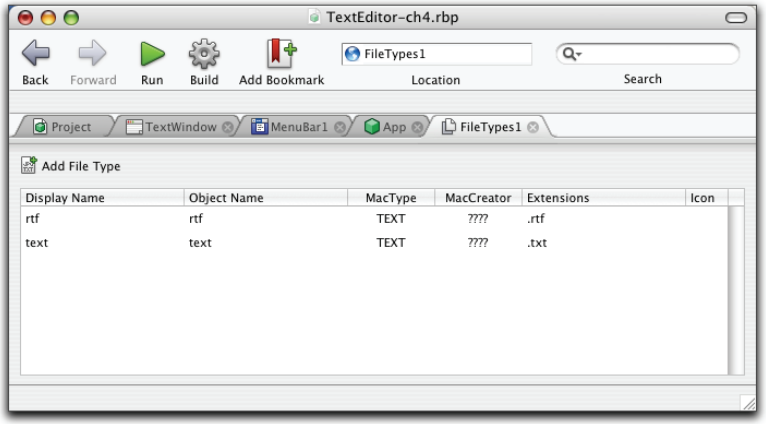
1. Click the Project tab to display the Project Editor and then choose Project ▶ Add ▶ File Type Set.
   REALbasic adds a project item called a File Type Set. It is a list of file type definitions that the application will understand.
2. Double-click the FileTypes1 project item to display its editor.
   REALbasic adds a Tab for the File Types Set.
3. Click the Add File Type button in the File Type Set toolbar to add a file type.
   A blank row is added to the list of file types. Since there are no file types yet, this list is blank.

Figure 26. The File Types Set Editor.



4. Enter **rtf** as both the Display Name and Object name, Mac Type of **TEXT**, Mac Creator of **????**, and Extension **.rtf** as shown in Figure 27. MacCreator and MacType are case-sensitive.
5. Click Add File Type again to add the text file type.
6. Enter **text** as both the Display Name and Object name, Mac Type of **TEXT**, Mac Creator of **????**, and Extension **.txt** as shown in Figure 27. MacCreator and MacType are case-sensitive.

Figure 27. The RTF and Text file types.



The Display Name is the name that the user sees in open-file dialog boxes and the Object Name is the name that you use in your code to refer to the file type.

The MacType and MacCreator codes are the four-character codes that Mac OS "classic" uses to identify files and applications. The MacCreator code of ???? tells TextEditor to open text files created by any application. If you entered a specific Creator code—such as "ttxt" or "R*ch"—TextEditor would only be able to open those text files.

File extensions are used by operating systems other than Mac OS "classic" to identify files. The "Icon" column enables you to specify a document icon for the file type. You would want to use this feature if your application stores documents using its own file type and you want documents belonging to the application to be recognized as such.

Adding a SaveFile Method

Next, you need to specify the actions to be taken when the Save menu item is chosen by the user. The Save menu item's event handler will call a separate stand-alone method rather than doing the save itself. The save will be done by the SaveFile method.

The reason we will do it this way is that we want to use the same method to manage both the Save and Save As menu items. You will add the Save As menu item in the section "Adding a Save As Menu Item" on page 44. The method will handle the following cases:

- The user chooses Save to save changes to an existing document.
- The user chooses Save or Save As to save an unsaved document or to save an existing document under a new name.

  In the latter case, the application must first present a save-file dialog box that lets the user enter a filename. In the former case, the application saves the document using the existing filename.

To do this, a value will be passed to the SaveFile method. If the value of True is passed, SaveFile will present a save-file dialog box. In this way, the same method can be called to manage either type of save.

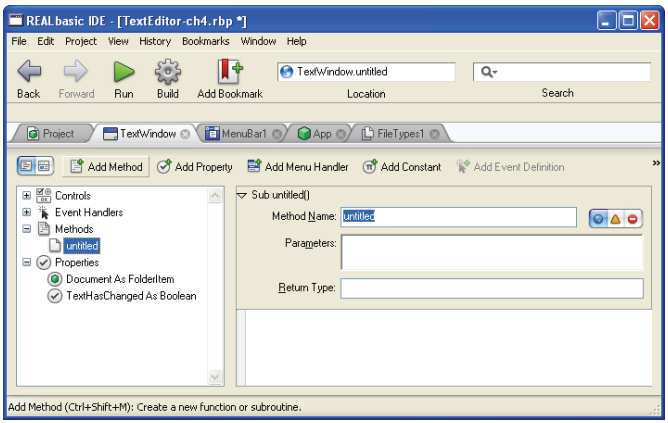
To add the SaveFile method, do this:

1 Click the TextWindow tab. If necessary, click the Code Editor icon to display its Code Editor.

2 Click the Add Method button in the Code Editor toolbar.

The Method declaration area appears above the code editing area. This is where you name the new method and declare parameters. Parameters contain data that is passed from the outside into the method. The method needs these values in order to do its job.

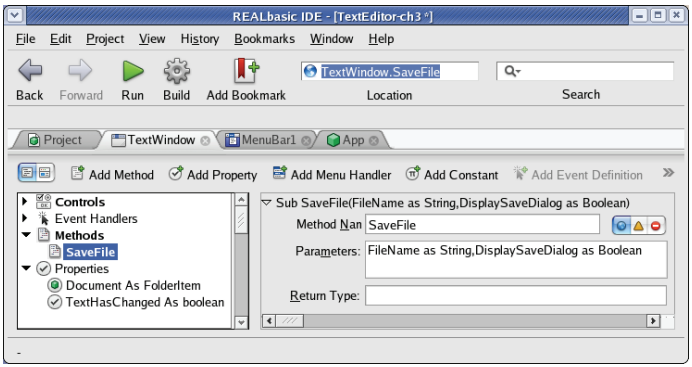
Figure 28. The Add Method declaration area.



The Add Method declaration area has fields for the name of the method, the parameters you must pass to the method when you call it, and the type of value that you return from the method. The last two items are optional. Parameters are used to pass information to a method from the outside. A method is not required to accept values from the outside via parameters. The Return Type is the output from the method. A method does not have to return a value; many methods do their jobs without having to return any value. This method has values that are input to the method but nothing is output.

3 Enter **SaveFile** as the method name and **FileName as String, DisplaySaveDialog As Boolean** in the Parameters area. Leave the Return Type area blank.

Figure 29. The SaveFile Method declaration.



The method name and parameters have been added to the heading area of the Code Editor area. If you need to change the name or parameters, you can open the declaration area of the Code Editor.

In the next step, you will enter the code that will handle the two cases we described.

4 Enter the following code for the SaveFile method into the Code Editor.

```
Dim f as FolderItem
If Document = Nil or DisplaySaveDialog then
 #If TargetWin32  //if on Windows
  f=GetSaveFolderItem("rtf",FileName) //use rtf file type
 #else  //on Linux or Macintosh
  f=GetSaveFolderItem("text",FileName) //use text file type
 #Endif
 If f <> Nil then //if the user clicked Save
  Title=f.Name //window Title gets document name
  Document=f  //window property gets folderitem
  End if
End if

If Document <> Nil then
 Document.SaveStyledEditField TextField
 TextHasChanged=False
End if
```

Remember to enter each printed line on a separate line in the Code Editor and do not split a long line into two lines. The characters "//" in a line indicate the everything after that is a comment and is not going to be executed. Comments appear in a different color in the Code Editor.

The logic of this method is as follows: If the Document property is undefined (i.e., its value is "Nil"), the document has not been saved, so the Save File dialog box

must be presented. The GetSaveFolderItem function does this. This is a method that is built into REALbasic.

If the application is running on Windows, it needs to save it as RTF; if it's on other platforms, it is saved as text. The special "#If…#Endif statement does this. The function TargetWin32 returns a value of True if the application is being compiled for any version of Windows; it returns False otherwise. That means that when the application is being compiled for Macintosh or Linux, it will include only the code in the #else statement.

For the Windows application, it includes the line:

```
f=GetSaveFolderItem("rtf",FileName)
```

but for Macintosh and Linux, it excludes that line and substitutes the following line:

```
f=GetSaveFolderItem("text",FileName)
```

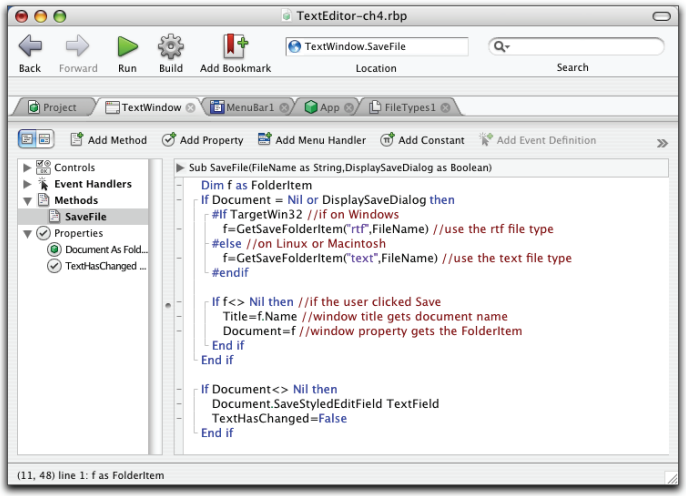
In this way, you can literally build separate versions of an application for each platform.

The line "Title=f.Name" sets the Title property of TextWindow to the text of the Name property of the FolderItem (i.e., the document). "Title" is a property of the Window class. Since TextWindow is a Window, it gets all the properties that belong to the Window class. The next line, "Document=f" sets the Document property of TextWindow to the opened document.

If the document exists (i.e., the FolderItem is not Nil), the Save As dialog box does not have to be presented; the user wants to resave an existing document under its current name. In this case, you use the SaveStyledEditField method of a FolderItem to save the contents of the TextField. "TextField" is the value of the parameter that is passed to the SaveStyledEditField method of the EditField class. We also reset the TextHasChanged Boolean property to False because the document has not changed since its last save.

The Code Editor should now look as shown in Figure 30 on page 42:

Figure 30. Code entered for SaveFile method.



We are not quite ready to use this method because we haven't added the line of code that sets the TextHasChanged property to True when the text of TextField changes. This will be done in next section.

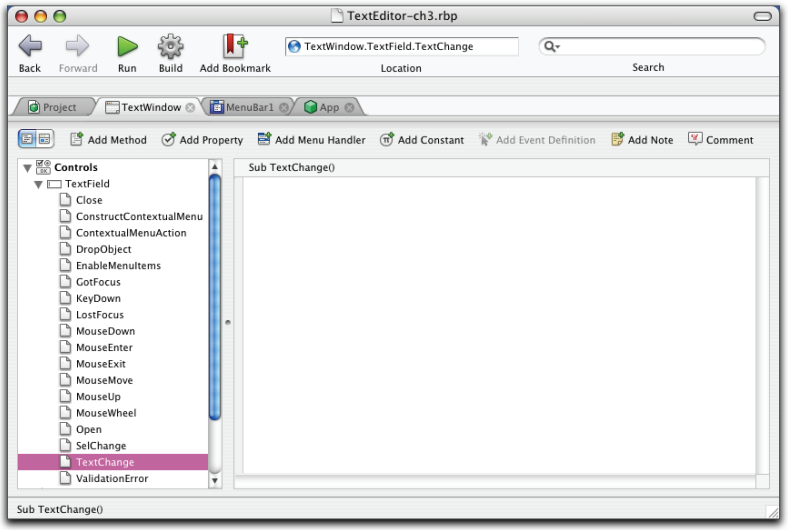
Managing the TextHasChanged Property

In order for the TextHasChanged property to be useful, it must be assigned a value of True whenever there is a change to the text in the EditField. This is done in the TextChange event handler of TextField.

An event handler runs automatically whenever a particular event occurs. Each REALbasic interface object comes equipped with a set of empty event handlers. These are events that REALbasic is capable of detecting automatically. You don't have to figure out when the event occurs, only what you want your program to do when it does occur. By adding code to an empty event handler, you specify what your application will do when a user interacts with the object in a certain way. This is the basic concept of *event-driven programming*.

To see which event handlers are available for an EditField, display the Code Editor of the TextWindow's Window Editor and expand the Controls item in the Browser. Then expand the TextField object. You will see a list of an EditField's event handlers. It will look like Figure 31. The one we need is the TextChange event handler. It runs whenever the text in the EditField changes.

Figure 31. TextField's Event Handlers.



To manage the TextHasChanged property, do this:

1. Highlight the TextChange event, as shown in Figure 31.
   The Code Editing area changes to show the code for this event handler. Of course, it is currently blank.
2. Add the following line of code to the event handler on the right of the divider:

```
TextHasChanged=True
```

Since you placed this line of code in the TextField's TextChange event handler, it will run whenever there is a change to the text in the TextField. REALbasic has the job of figuring out when the text has changed.

In the Online Language Reference, the event handlers that are available for each control are described in the Events table for that control.

Handling the Menu Item

The menu handler for the Save menu item calls the SaveFile method that you just wrote and passes the value of False to the DisplaySaveDialog parameter to prevent the method from displaying the save-file dialog box (unless it is an unsaved document). The menu handler belongs in the Code Editor for TextWindow (rather than the App class) since it will save the contents of a particular document window.

To handle the menu item, do this:

1. Display the Code Editor for TextWindow and click the Add Menu Handler button in the Code Editor toolbar.
   A new menu handler is added to the browser area and a new declaration area appears in the Code Editing area.
2. Choose FileSave from the MenuItem Name drop-down list and press Tab.

REALbasic changes the name of the new menu handler from Untitled to FileSave in the browser and declaration areas.

3 Enter the following code:

```
SaveFile Title, False
```

This menu handler calls another method, SaveFile, that does the work. The items that follow this call —Title, and False —are the values of the parameters that are passed to the SaveFile method.

Remember that SaveFile takes two parameters. The first is a String variable and is the default name of the file to be saved. The second is a Boolean variable that tells SaveFile whether it needs to display a "save changes" dialog box. The term "Title" is the Title property of TextWindow—the text that appears in the TextWindow's Title bar. It's used as the default name.

The value of False is passed as the value of the DisplaySaveDialog parameter.

When you run the application and save the document the first time, the default text will be "Untitled," since that is the default window title.

4 Save your project.

5 Click the Run in the Main Toolbar to test your application.
Notice that the Save menu item is disabled initially.

6 Type some text. Use the Save menu item to save the document.
Notice that the Save menu item becomes disabled until you modify the text in the text editor.

7 Quit out of your application to return to the REALbasic IDE.

Adding a Save As Menu Item

A Save As menu item performs the same function as a Save menu item, except that it always presents a save-file dialog box that allows the user to save the existing document under a new name. It is implemented in the same fashion as the Save menu item.

To add the Save As menu item, do this:

1 Click the MenuBar1 tab in the Tab bar to display its Menu Editor.

2 Click on the File menu in the Menu Editor to display it.

3 Click the Add Menu Item button in the Menu Editor's Editor toolbar to add a new menu item to the File menu.
REALbasic adds a new menu item to the File menu named "untitled."

4 Using the new menu item's Properties pane, change its Text property to **Save As...**.Use three periods instead of the ellipsis character (which consists of three dots).

5 Drag it vertically between the Save and Exit menu items (Windows and Linux) or the Save and Quit menu items on Mac OS X.

6 Click on the TextWindow tab to display TextWindow's editor.

7 Click on the Code Editor icon in the Editor toolbar if the Code Editor is not already displayed.

8 Click the Add Menu Handler button.

REALbasic displays a new menu handler declaration area.

9 Choose the FileSaveAs item from the MenuItem Name drop-down list and press Tab.

10 Enter the following code in the FileSaveAs menu handler:

```
SaveFile Title, True
```

This menu handler manages the save using the SaveFile method but forces the save-file dialog box to be presented because True is passed as the second parameter.

11 Save your project and test the Save and Save As commands by clicking the Run button in the Main Toolbar. Enter some text and save it using the Save command. Then try the Save As command. Notice that the Save As command uses the existing window title as the default document name.

12 When you are finished, quit out of the test application to return to the REALbasic IDE.

Adding an Open Menu Item

Now that you have implemented the Save and Save As menu items, the user needs to be able to open any of the documents that he has saved. The following exercise implements an Open menu item. You will:

- Add the Open menu item,
- Write a menu handler for the item.

The Open menu item should be enabled all the time since a user should be able to open an existing document even when no windows are open. This option is available on Macintosh versions of the application. Therefore you should use the App object to manage the Open menu item.

Creating the Open Menu Item

To create the Open menu item, do this:

1 Click on MenuBar1 tab in the Tab bar and click on the File menu in the Menu Editor to display its items.

NOTE: If the Menu Editor does not open, check to see if the Run tab is in the Tab bar. If it is, close it.

2 Click the Add Menu Item button in the Menu Editor toolbar to add a new menu item to the File menu.

3 Use the new menu item's Properties pane to change its Text property to **&Open...**, enter **O** in the Key property area, and check the MenuModifier checkbox.
On Windows and Linux versions, the "O" in Open is underlined, indicating that it is the keyboard accelerator. If you are using Mac OS X, you can preview the menus for Windows and Linux by clicking the Preview Mode icon for those platforms.

4 In the Menu Editor, drag the Open menu item vertically between the New and Save menu items.

5 Click on the Open... menu item and then click the Add Separator button in the Menu Editor toolbar.
REALbasic places a separator between the Open... and Save menu items.

Handling the Menu Item

Like the New menu item, the Open menu item should be available even if there are no open document windows. This case occurs on the Macintosh platform when the user closes all the open windows but doesn't quit out of the application. That means that its menu handler belongs in the App object.

To handle the Open File menu item, do this:

1 Click the App object's tab in the Tab bar or, if it is not open, double-click the App object in the Project Editor to open its Code Editor.
The Code Editor for the App object appears.

2 Click the Add Menu Handler button to create a menu handler for the Open menu.

3 Choose the FileOpen menu handler from the MenuItem Name drop-down list and press Tab.
REALbasic adds the FileOpen item to the list of menu handlers in the browser area.

4 Enter the following code into the FileOpen menu handler in the Code Editor:

```
Dim f as FolderItem
Dim w as TextWindow
#If TargetWin32 then //Windows only
 f=GetOpenFolderItem("text;rtf") //displays open-file dialog
#else //Linux and Macintosh
 f=GetOpenFolderItem("text") //displays open file-dialog
#endif
If f <> Nil then //the user clicked Open
 w=New TextWindow  //create new instance of TextWindow
 f.OpenStyledEditField w.TextField
 w.Document=f  //assign f to document property of TextWindow
 w.title=f.Name  //assign name of f to Title property of w
End if
```

The first two lines of this method create a new FolderItem object—a reference to a document—and a new instance of the TextWindow class to display the document. At this point, f contains no value, it is just a container that is capable of referring to

a document. Similarly, the object "w" doesn't actually refer to a new instance of TextWindow until the New function creates it.

The method then calls the global method GetOpenFolderItem. It displays the open-file dialog box and allows the user to select only the file types that are passed to it.

On the Windows platform, the Text editor will use the rtf format to store styled text documents, so it should be able to open both styled and plain text documents. The parameter ("text;rtf") instructs the GetOpenFolderItem method to display only text or rtf documents.

On Macintosh and Linux, the styled text documents will be stored using the .txt file format anyway, so rtf files should not be presented to the user in the open-file dialog.

In case of difficulty

Be sure to separate "text" and "rtf" by a semicolon, not a comma in the GetOpenFolderItem line. If you use a comma, REALbasic will think that you are passing two separate parameters. The GetOpenFolderItem method is expecting only one parameter, a list of valid file types. Also, enclose the whole string in quote, not each file type.

If the user successfully opens a text document, the GetOpenFolderItem function returns a reference to the document in the FolderItem object, f. The code first tests whether f is still Nil—it would be Nil if the user clicked the Cancel button in the open-file dialog—before creating a new window for the document and calling the OpenStyledEditField method of the FolderItem class. This method places the text that is now in f into the TextField belonging to the new instance of TextWindow.

The menu handler then sets the Document property of TextWindow to the FolderItem (the document the user selected) and sets the title of the window to the name of the document.

NOTE: If you are confused about how the various calls to built-in methods work, consult the online reference entries for FolderItem, GetOpenFolderItem, and the Window class.

5. Save your project.
6. Click the Run button in the Main Toolbar and try out the Open and Save commands.

   NOTE: If you have "Unknown Identifier" errors or some other problems, double-check to make sure that you have renamed all objects and have placed your code in the correct Code Editor. If you still have trouble locating the problem, open the TextEditor-ch4 project on your REALbasic CD and compare your project to that one.
7. When you are finished, quit the application to return to the REALbasic IDE.

Review

In this chapter you learned how to add the capability to open, create, close, and save documents in your application.

To Learn More About:	Go to:
REALbasic Files	REALbasic User's Guide: Chapters 6, 7, 8.
REALbasic commands and language	REALbasic Language Reference

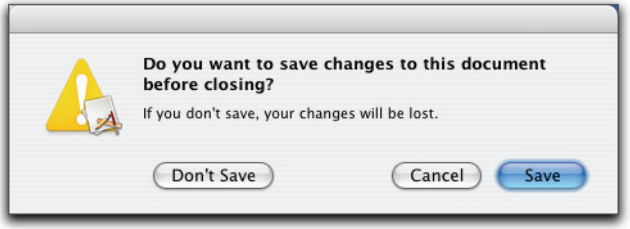
CHAPTER 5

Adding a "Save Changes" Dialog Box

In "well-behaved" applications, the application gives the user a chance to save changes to open documents whenever he closes a window or quits the application with unsaved changes. This application is no different.

In this chapter, you will create, install, and activate the Save Changes dialog box shown in Figure 32.

Figure 32. The Save Changes dialog box.



This dialog box will appear when the user closes a document window with unsaved changes or quits the application when a document with unsaved changes is open.

Getting Started

If the TextEditor project is not already open, locate the REALbasic project file that you saved at the end of last chapter ("TextEditor-ch4"). Launch REALbasic and open the project file. If you need to, you can use the "TextEditor-ch4"file that is in the Tutorial Files folder on the REALbasic CD.

Creating the Dialog Box

In this chapter you will create the dialog box using a built-in REALbasic class, the MessageDialog class. With the MessageDialog class, you can create dialog boxes that display a message and accept user feedback without explicitly creating a window and designing it in a Window Editor. (You will learn how to create a second window and manage communications with the main text editor window in Chapter 9.)

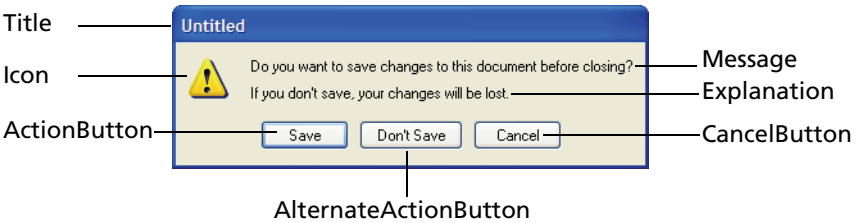
The MessageDialog class does not use the Window Editor. Instead, you create an instance of the MessageDialog class in your code and use its properties to customize its appearance. A MessageDialog window can contain the following elements:

Property	Description
Icon	Icon to be displayed on the left side of the dialog.
Message	Text of the main message displayed in the dialog.
Explanation	Secondary explanatory text to be displayed below the Message.
ActionButton	The default "Accept" button for the dialog.
CancelButton	The button for canceling out of the dialog.
AlternateActionButton	An alternate "Accept" button for an alternate action.
Title	The text in the Title bar of the dialog. Not shown on Macintosh.

The ActionButton is shown by default; you show other elements by assigning values to their properties. The MessageDialog class manages the sizing of the dialog window and the placement of the elements automatically.

In this case, these properties are configured as shown in Figure 33.

Figure 33. The a MessageDialog box, with its properties shown.



The Macintosh version of the MessageDialog box places the buttons in different locations than the Windows and Linux versions and does not support the Title

property. Otherwise, the dialog box shown in Figure 33 is identical to the one shown in Figure 32 on page 49.

Each of the buttons in a MessageDialog window is a MessageDialogButton object. A MessageDialogButton has properties of its own, shown in the following table:

Property	Description
Cancel	If True, the button will respond to the Esc key or, on Macintosh, the Command-period sequence. The Cancel property can be True on only one button.
Caption	The button's text.
Default	If True, the button is highlighted as the default button in the dialog box. The Default button will respond to the Enter and Return keys.
Visible	If True, the button will be shown in the dialog box. Only the ActionButton is shown by default.

To create the dialog box, we need to write a method that creates an instance of the MessageDialog class, assigns values to its properties, and takes an appropriate action depending on which button the user clicks. You then display the MessageDialog box by calling its ShowModal method.

Specifying the MessageDialog's Properties

By default, only the ActionButton is actually displayed. To display the other two buttons, you must set their Visible properties to True. We also need to specify the Icon to be displayed (the choices are Note icon, Stop icon, Warning icon, or no icon). With that in mind, the following code specifies the design of the Save Changes dialog box:

```
Dim d as New MessageDialog  //declare the MessageDialog object
Dim b as New MessageDialogButton //for handling the button the user pushed
d.icon=MessageDialog.GraphicCaution //display the warning icon
d.ActionButton.Caption="Save"
d.CancelButton.Visible=True     //show the Cancel button
d.AlternateActionButton.Visible=True   //show the "Don't Save" button
d.AlternateActionButton.Caption="Don't Save"
d.Message="Do you want to save changes to this document before"_
          +" closing?"
d.Explanation="If you don't save, your changes will be lost"
```

There are some interesting new features about REALbasic syntax that are illustrated here. First, the "dot" notation that we used to refer to a property of an object is extended to refer to the properties of one object that belong to another object. For example, the line:

```
d.ActionButton.Caption="Save"
```

The ActionButton is a property of the MessageDialog object, d, and Caption is a property of ActionButton. REALbasic handles this situation simply by extending

the "dot" notation to follow the hierarchy. The "dot" notation works like a possessive in a spoken language. In this case, the Caption property belongs to the ActionButton and the ActionButton belongs to the MessageDialog, d.

Second, the long string that is assigned to the Message property is broken up into two lines.

```
d.Message="Do you want to save changes to this document before"_
        +" closing?"
```

It uses the underscore keyword to tell REALbasic that the next line in the Code Editor is actually a continuation of the first line. If you enter these two lines just as shown here, REALbasic will put the two lines together to form one logical line. It's equivalent to typing the entire string in one line in the Code Editor.

Third, the code uses a class constant belonging to the MessageDialog class to specify the icon to display in the dialog. The Icon property is actually an integer and you can specify the icon using the integers 0 to 3 (or -1 for no icon), but it your code will be much more readable if you use a constant belonging to the class to specify the value.

There are four possible types of icons to show: Note, Caution, Stop, and Question. You can specify the one you want using the class constants GraphicNote, GraphicCaution, GraphicStop, and GraphicQuestion. Or, you can use GraphicNone to specify no icon. You use the "dot" notation, so the line:

```
d.icon=MessageDialog.GraphicCaution
```

specifies the caution icon.

Managing User Input

After we construct the MessageDialog, we need to display it to the user, get his response, and take appropriate action. That's handled by the next section of code. The variable "b" is declared as a MessageDialogButton and it will contain the button that the user has clicked.

```
b=d.ShowModal    //display the dialog & wait for user input

Select Case b //determine which button was pressed
  Case d.ActionButton //user pressed Save
    SaveFile Title,False  //call the SaveFile method to save it
  Case d.AlternateActionButton
    //user pressed Don't Save
    //allow the close without saving changes
  Case d.CancelButton //user pressed Cancel
    Return True //cancel the window close
End Select
```

The ShowModal method of the MessageDialog class returns a MessageDialogButton (which we declared in the first code segment). This MessageDialogButton is the one that the user clicked. Code execution stops at that line until the user clicks one of the buttons. So, we need to figure out which one it is. That's what the Select Case statement does.

Each Case statement in the Select Case…End Select structure tests for a particular value of the MessageDialogButton object, b. There are only three possibilities. If it is the ActionButton, it calls the SaveFile method to save the contents of the window, if it is the AlternateActionButton it does nothing, and if it is the Cancel button, it blocks closing the window or quitting the application by returning the value of True from the CancelClose event handler. This puts away the MessageDialog without closing the text window.

Adding the Save Changes Dialog Box to the Project

The final step is to add code that displays the SaveChanges dialog box when the user chooses the Exit menu item (Quit on Macintosh) or closes a window and there are unsaved changes to the contents of an open window. This is done in the CancelClose event handler of TextWindow.

The Exit (or Quit) menu item is different from the menu items that you have added in several ways. First, it is an instance of the QuitMenuItem class, rather than the MenuItem class. You can verify this by highlighting the Quit menu item in the Menu Editor and checking its properties.

Second, the Quit menu item is enabled by default. You have been able to use the Quit menu item to return to the Development environment even though you have not enabled it.

Finally, the Quit menu item also has its own menu handler — it calls the built-in Quit method. This method tries to quit the application. If any windows are open, it calls each window's CancelClose event handler. This event handler gives you a chance to cancel the quit or perform actions prior to the quit.

If the CancelClose event handler returns False (the default action) then the window's Close event handler will be executed. It means, "Don't cancel the close." If the CancelClose event handler returns True, REALbasic stops sending CancelClose or Close events. This blocks the close or the quit, if the user chose the Quit menu item.

The CancelClose method that you will write displays the Save Changes dialog box if the TextHasChanged property is True. It then determines which button in the dialog box the user has clicked. Only if the user clicks the Save button is the SaveFile method called.

To add the CancelClose code, do this:

1. If it not already displayed, click the TextWindow tab to display its Window Editor and then click the Code Editor icon in the Editor Toolbar.
2. In the Code Editor for TextWindow, expand the Event Handlers item and highlight the CancelClose event.
3. Enter the following code:

```
Dim d as New MessageDialog  //declare the MessageDialog object
Dim b as New MessageDialogButton //for handling the result
If TextHasChanged then
 d.icon=MessageDialog.GraphicCaution //display warning icon
 d.ActionButton.Caption="Save"
 d.CancelButton.Visible=True     //show the Cancel button
 d.AlternateActionButton.Visible=True   //show the "Don't Save" button
 d.AlternateActionButton.Caption="Don't Save"
 d.Message="Do you want to save changes to this document"_
          +" before closing?"
 d.Explanation="If you don't save, your changes will be lost. "

 b=d.ShowModal     //display the dialog
 Select Case b   //determine which button was pressed
  Case d.ActionButton   //user clicked Save
  SaveFile Title,False
  Close
 Case d.AlternateActionButton //user clicked Don't Save
     //allow the close without saving changes
 Case d.CancelButton //user clicked Cancel
 Return True //cancel the window close
 End Select
End if
```

The code tests whether the text has changed by testing the value of the TextHasChanged property, and, if it has, it displays the Save Changes dialog box.

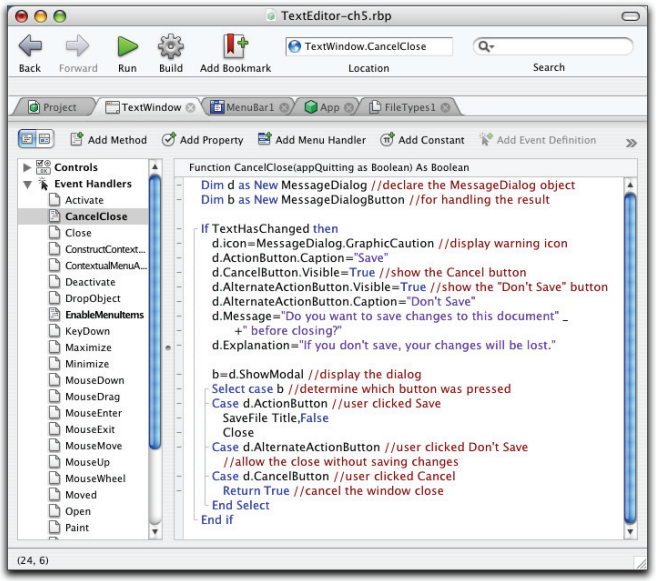
The d.ShowModal statement calls the ShowModal method of the MessageDialog class and returns the MessageDialogButton that the user clicked.

The Select Case structure determines which button the user clicked. If the Don't Save button was clicked, the quit continues without saving the document because CancelClose returns False and no method for saving the document is called. If the user clicks Cancel, the CancelClose event handler returns True, canceling the close/quit. If the user clicks Save, the SaveFile method runs. Its parameters are the Title of the window (the Title property of the Window class contains the title of the window instance) and False—telling SaveFile not to display the Save File dialog

box. The Close method of the Window class is needed to close the window if multiple unsaved windows are open.

4 Save your project as **TextEditor-ch5**.
The Code Editor for CancelClose should now look like Figure 34.

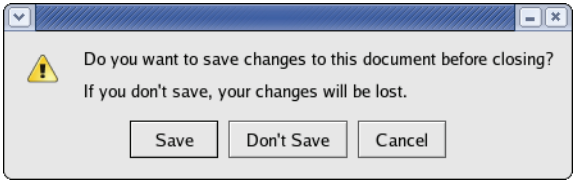
Figure 34. The CancelClose event handler.



5 Test the SaveChanges dialog box by switching to the Debugging environment (click the Run button in the Main Toolbar), creating a new document, and entering some text. Then close the window or quit the application.

The Save Changes dialog box should appear when you close the window or choose Quit.

Figure 35. The SaveChanges dialog box.



If you click Don't Save, the application will go ahead and close; if you click Save, a save-file dialog box will appear before the close, and if you click Cancel, the close operation will be aborted.

Review

In this chapter you learned how to add the a new window to the application.

To Learn More About:	Go to:
REALbasic Windows and Controls	REALbasic User's Guide: Chapter 3.
REALbasic commands and language	REALbasic Language Reference

CHAPTER 6

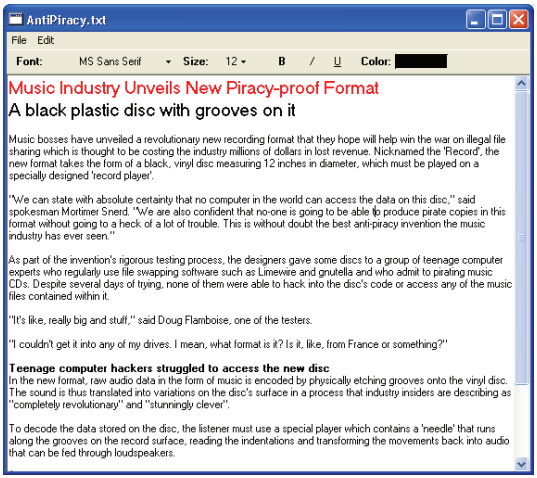
Working with Styled Text

In this chapter you will work with the styled text features in REALbasic. You will implement font size, font style, and color controls. Your code will also place check marks next to the currently selected style and font size so that the user knows the current settings.

In the next chapter you will add a Font menu that allows the user to use any font installed on his computer.

When you are finished, a document window will look as shown in Figure 36.

Figure 36. The finished TextEditor application window.



As you can see, the controls appear inside the window. The Font and Font Size controls are pop-up menus, the Style controls are buttons, and the color control is a Canvas control that indicates the color of the selected text and displays the Color Picker when the user clicks it.

Getting Started

If it isn't already open, locate the REALbasic project file that you saved at the end of last chapter ("TextEditor-ch5"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch6" that is in the Tutorial Files folder on the REALbasic CD.

Configuring TextField for Styled Text

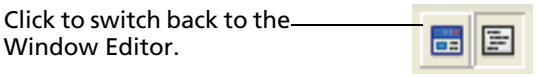
Before implementing the Style and Size menus, we need to move the top of the TextField down a few pixels to make room for the controls.

To configure the TextField for styled text, do this:

1 Click the TextWindow tab in the Tab bar to display the Window Editor Layout view, or, if it is not already shown, double-click the TextWindow item in the Project Editor.

2 Switch from TextWindow's Code Editor to its Window Editor by clicking on the Window Editor's Edit Mode icon.

Figure 37. The Edit Mode icons.



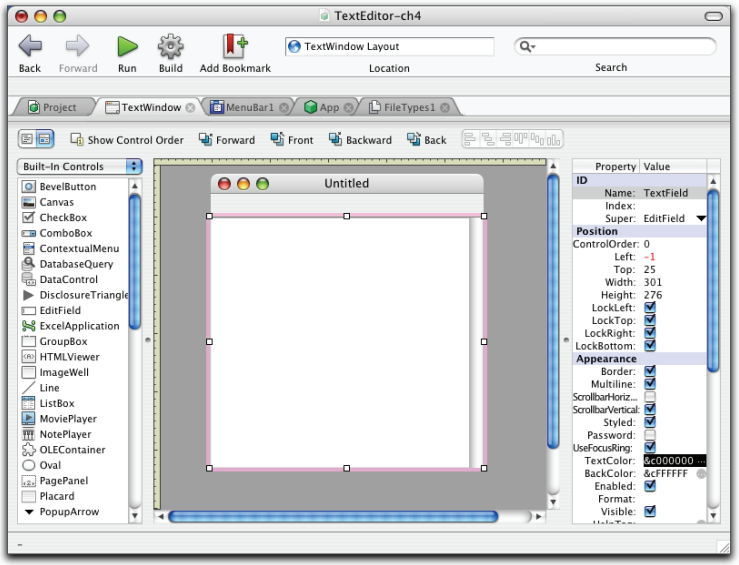
3 Click once on the surface of TextField to select it.

The Properties pane changes to show TextField's properties.

4 Set the Top property of TextField to **25**.

5 Reduce the value of the Height property by 26.
Your TextWindow should now look like Figure 38.

Figure 38. The TextWindow after adding space for the Font and Style controls.



6 Save the project as **TextEditor-ch6**.

Creating the Font Size Pop-up Menu

In this section you will create a Size menu and its menu items.

Creating the Size Menu and its Menu Items

In this exercise, you will create a Size menu and menu items corresponding to the font sizes of 9, 10, 12,14, 18, 24, and 36.

To create the Size menu and its label, do this:

1 Drag a StaticText control Aa from the Controls list to the top area of TextWindow, in the empty space above TextField.
It will serve as the label for the Size menu. When it is selected in the Window Editor, the Properties pane shows its properties.

2 Using the Properties pane for the StaticText control, set its properties as shown in the following table. You can use the Tab key to move from field to field.

Table 1: Properties of the StaticText Control.

Property	Value
Left	190
Top	4
Width	29
Height	16
Text	Size:
TextAlign	Right
TextFont	System
TextSize	0
Bold	Checked

You may be wondering why you set the TextSize property to zero. This is the property that controls the font size of the StaticText's visible text. You can either set TextSize to a particular font size or use zero to tell REALbasic to pick the default font size for the platform on which the application is currently running. Since the best font size usually differs for by platform, this option is provided so that you can easily get an attractive font size without adding code that specifies a different font size for each platform.

Likewise, the TextFont, "System," does not refer to a particular font. Rather, it tells REALbasic to chose the system font for the platform on which the application is running.

If your application will be used on only one operating system, you can just enter a specific font size for the TextSize property.

3 Next, drag a BevelButton control to the right of the StaticText control.
The BevelButton is an especially versatile type of control that can be configured either as a pushbutton or pop-up menu and can display either text or a picture (or both).

4 Using the Properties pane for the BevelButton, set its properties as shown in Table 2.

Table 2: Properties of the Size Menu Control.

Property	Value
Name	SizeMenu
Left	221
Top	4
Width	43
Height	16
Caption	(set to blank)

Table 2: Properties of the Size Menu Control.

Property	Value
CaptionAlign	Center
CaptionPlacement	Normally
HasMenu	Normal Menu
TextFont	System
TextSize	0

The HasMenu property instructs the BevelButton to behave like a popup menu rather than a button.

The next task is to create the menu items. When the application runs, this is done at the time the window opens.

5 Switch to TextWindow's Code Editor and open the SizeMenu item in the Controls category and highlight the Open event handler.

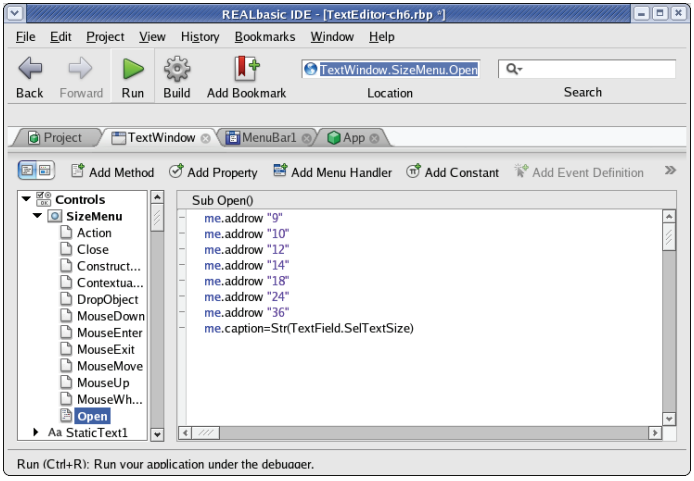
The Open event runs when the window containing the control opens.

6 Enter the following code into this event handler:

```
me.addrow "9"
me.addrow "10"
me.addrow "12"
me.addrow "14"
me.addrow "18"
me.addrow "24"
me.addrow "36"
me.caption=Str(TextField.SelTextSize)
```

Your code editor should look like this.

Figure 39. The SizeMenu's Open event handler.



Using the Properties pane, we declared that the Bevelbutton will behave as a pop-up menu but we didn't supply it with any menu items. The Open event handler takes care of this.

The first seven lines call the AddRow method of the BevelButton class. This method adds an item to its pop-up menu. The last line sets the default value of the Caption property to the font size at the text insertion point when the window first appears. The SelTextSize property of TextField is a function that returns the font size at the insertion point in TextField. It returns the font size as an integer number. This means that the selected menu item will agree with the font size of the selected text in TextField.

The Str function converts the (integer) font size of the selected text to a string. You need to do this conversion because the Caption property accepts only strings. If you try to pass it a numeric value, you will get an error message when you try to test the application.

The term "Me" refers to the event handler's control, SizeMenu. You could have also written "SizeMenu.addrow", and so forth, but when you use Me, the code is generic and can be pasted into another BevelButton control and it will work without modification. It will also continue to work if you change the name of the control.

Finally, we need to tell REALbasic what to do when the user selects a menu item. We do this in SizeMenu's Action event handler. It runs when the user makes a selection from the menu.

7 Click on SizeMenu's Action event handler and enter the following code:

```
Me.Caption=Me.List(Me.MenuValue)
TextField.SelTextSize=Val(Me.Caption)
```

The MenuValue property is the *number* of the selected menu item and the List method returns the text of the menu item corresponding to the number passed to it. That is, the first line sets the Caption property (the text displayed by the control) to the font size that the user chooses.

The second line assigns the selected font size (i.e., the Caption property converted to a number) to the SelTextSize property of the TextField. The Val function converts a string to a number (assuming the string spells out a number). SelTextSize is the font size of the selected text. If no text is selected, any text that the user types is in the SelTextSize font size.

Trying out the Size Menu

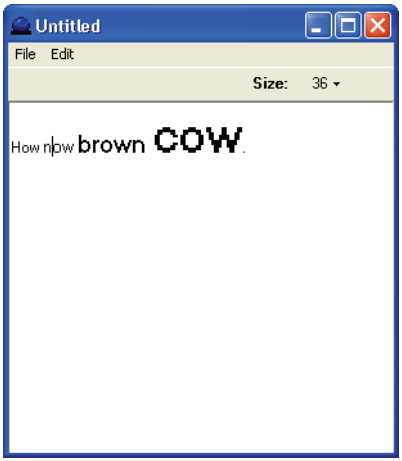
Save your project and try it out (click Run in the Main Toolbar). Type a few words and try changing the font size. Hey, it works!

An Unresolved Issue

If you assign different font sizes to different words and then move the insertion point from word to word, you'll notice that the Size menu doesn't indicate the current font size. That is, the Size menu can talk to TextField, but it doesn't get any

feedback from TextField about the font size of the currently selected text. The problem is illustrated in Figure 40 on page 63.

Figure 40. The Size Menu Isn't Being Updated.



In Figure 40, the Size menu says "36" because I just finished setting the word "cow" in 36 point type. But when I moved the insertion point into the word "now", which is in 10 point, the Size menu still says "36." This is no good.

Updating the Font Size Menu

To update the Font Size menu, use the SelChange event of TextField. SelChange runs whenever the user changes the text selection. This includes moving the insertion point to different text without selecting a group of characters.

1. In TextWindow's Code Editor, open the TextField control in the Controls group and highlight the SelChange event handler.
2. Enter the following code into this event handler:

```
//update Font Size menu
If Str(Me.SelTextSize) <> SizeMenu.Caption then
  SizeMenu.Caption=Str(Me.SelTextSize)
  SizeMenu.MenuValue=SetFontSizeMenu(Me.SelTextSize)
End if
```

This event handler uses an If statement. It checks to see whether the font size of the selected text is the same as the current setting of the SizeMenu. If not, it resets the Caption and MenuValue properties of SizeMenu. To do the latter, it needs a function that converts the font size of the selected text to the menu item number (the MenuValue property goes from zero to 6; it doesn't contain the text of the menu item). The SetFontSizeMenu method is a very simple function that does this conversion. You need to add it to the project now.

3 With the Code Editor for TextWindow displayed, click the Add Method button in the Code Editor toolbar.

The Method declaration area appears above the code editing area.

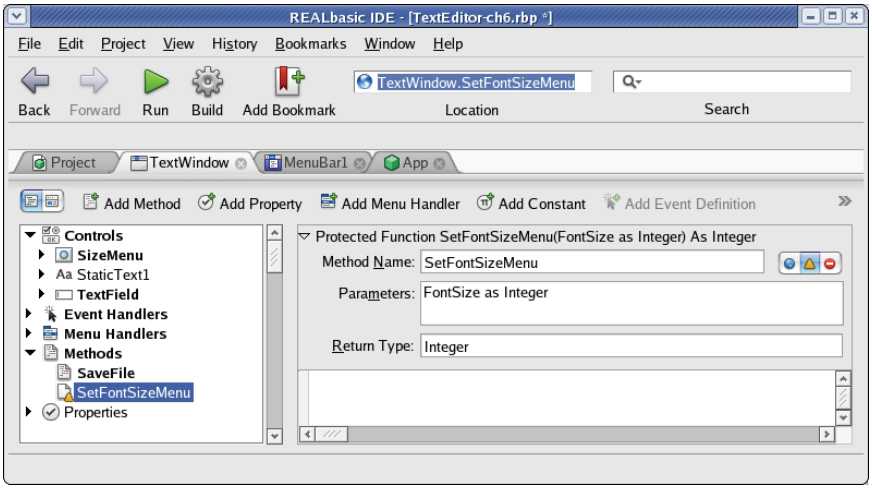
4 Enter the name **SetFontSizeMenu**, parameter **FontSize as Integer**, and Return type of **Integer**.

5 In the trio of icons to the right of the method declaration, click the middle icon.

This choice sets the *Access Scope* for the method to Protected. This means that the method can be called only from code belonging to TextWindow or any of its controls, not from code that's outside of TextWindow.

The dialog box should now look like Figure 41.

Figure 41. SetFontSizeMenu's parameters.



In the Methods group in the Browser area, SetFontSizeMenu has a small badge on its icon, indicating that it is a Protected method.

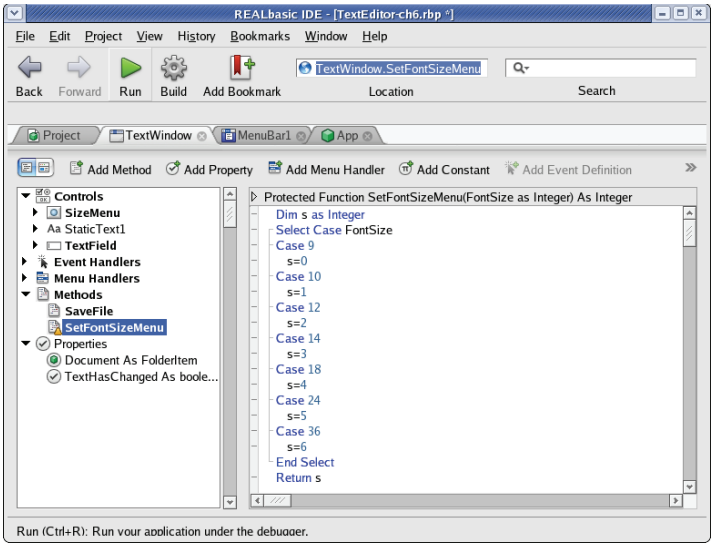
6 Enter the following code into the Code Editor for SetFontSizeMenu:

```
Dim s as Integer
Select Case FontSize
  Case 9
    s=0
  case 10
    s=1
  case 12
    s=2
  case 14
    s=3
  case 18
    s=4
  case 24
    s=5
  case 36
    s=6
End Select
Return s
```

The Select Case statement takes the parameter passed to method (the current font size) and chooses the corresponding sequential menu item number. The Return statement returns the contents of the variable s after it has been set by the Select Case statement.

Your Code Editor should now look like this.

Figure 42. The SetFontSizeMenu method.



7 Click the Run button in the Main Toolbar to try out the application.
As you move the insertion point to text of different font sizes, the Size menu updates automatically.

Implementing the Font Style Controls

The next task is to add the three buttons to the right of the Font Size menu in Figure 36 on page 58 that allow the user to apply the Bold, Italic, and Underline styles.

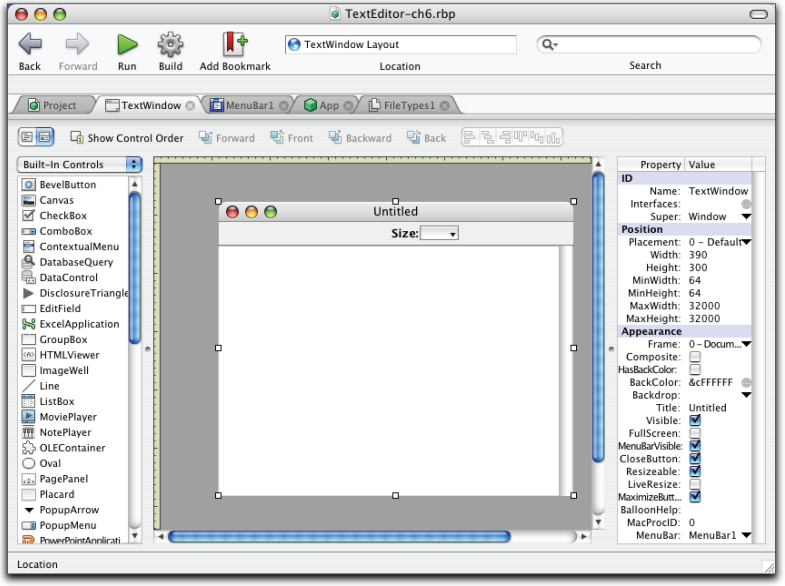
We will also use BevelButtons to control font style. This time, however, they will be used as buttons rather than pop-up menus.

Creating the Style Buttons

To create the Style buttons, do this:

1 In TextWindow's Window Editor, enlarge TextWindow by dragging its Grow handle to the right to make room for the additional controls.
The Grow handle is the bottom-right corner of the window. You may need to enlarge the IDE window itself by dragging its Grow handle and possibly increasing the size of the Window Editor area by dragging the divider that separates it from the Properties pane to the right After the adjustment, the Window Editor should look something like this:

Figure 43. The Window Editor after enlarging the window.



2 Click once on the Bevelbutton control and then drag a small square region to the right of the Font menu, as shown here.

Figure 44. Creating a control by dragging from top-left to bottom-right.

When you release the mouse button, the BevelButton control will be created in the size, shape, and location that you specify. This is closer to its final shape than the default size and shape. This is a great way to add a control when you don't want to use its default size and/or shape.

3 Using its Properties pane, set its properties as follows:

Table 3: Properties of the BevelButton Bold Control.

Property	Value
Name	BoldButton
Left	289
Top	4
Width	16
Height	16
Caption	B
CaptionAlign	Center
TextFont	System
TextSize	9
Bold	True (checked)
Italic	False
Underline	False
ButtonType	Toggles

4 Duplicate the Bold button twice (press Ctrl+D or ⌘-D on Macintosh), move the new buttons to the right of the Bold buttons—to the approximate positions of the Italic and Underline buttons in Figure 36 on page 58, in the order Bold, Italic, Underline — and use the Properties pane to set their properties as follows:

Table 4: Properties of the Italic and Underline Controls.

Property	Italic Button	Underline Button
Name	ItalicButton	UnderlineButton
Left	319	349
Top	4	4
Width	16	16
Height	16	16
Caption	I	U
CaptionAlign	Center	Center

Table 4: Properties of the Italic and Underline Controls.

Property	Italic Button	Underline Button
HasMenu	No Menu	No Menu
TextFont	System	System
TextSize	9	9
Bold	False	False
Italic	True (checked)	False
Underline	False	True (checked)
ButtonType	Toggles	Toggles

Next, we need to add the code that tells REALbasic what to do when the user clicks each button. To do this, we use the Action event handler for each button. It runs whenever the button is clicked.

5 In the Code Editor for TextWindow, highlight the Action event handler for BoldButton and add the following line of code:

```
TextField.ToggleSelectionBold
```

This line reverses the Bold attribute of selected text in TextField. If the current text is bold, it removes the Bold attribute; if the text isn't bold, it adds it.

6 Highlight the Action event handlers for ItalicButton and UnderlineButton and add **TextField.ToggleSelectionItalic** to ItalicButton and **TextField.ToggleSelectionUnderline** to UnderlineButton.

7 Save your project.

Updating the Style Controls

The final step is to add code that updates the Bold, Italic, and Underline buttons when the user moves the insertion point to text that has different style attributes.

To do this, you will add additional code to the SelChange event handler of the TextField. This event runs when the text selection has changed.

To update the style buttons, do this:

1 In the Code Editor for TextWindow, expand the TextField item.

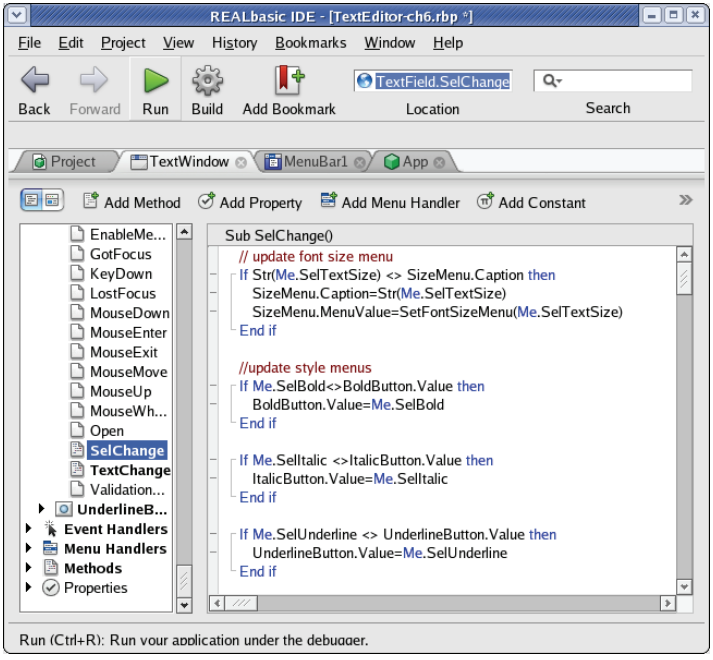
2 Click on the SelChange event handler and add the following code below the existing code:

```
//update style buttons
If Me.Selbold <> BoldButton.Value then
  BoldButton.Value=Me.Selbold
End if
If Me.Selitalic <> ItalicButton.Value then
  ItalicButton.Value=Me.Selitalic
End if
If Me.Selunderline <> UnderlineButton.Value then
  UnderlineButton.Value=Me.Selunderline
End if
```

Each "If" statement checks to see whether the value of a button (i.e., whether it is on or off) matches a style attribute of the selected text. If not, it sets the button's value to the state of that style for the selected text. For example, if BoldButton's Value property is True, but the selected text is not bold, the code sets BoldButton's Value property to False.

The SelChange event handler should now look like Figure 45.

Figure 45. The SelChange event handler after adding code for the Style controls.



Testing the Style and Size Controls

Now that all the code is in place, you are ready to see how it works.

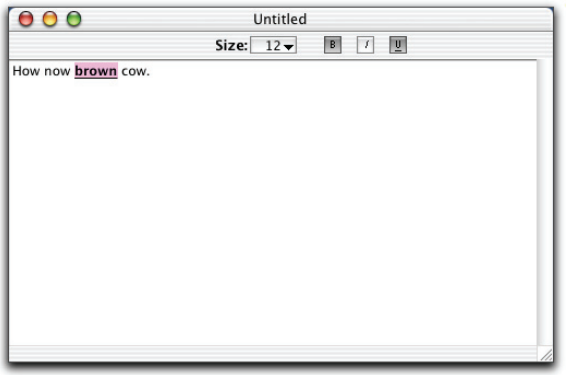
To use the styled text editor, do this:

1. Click the Run button in the Main Toolbar.
2. When the application launches, enter some text in the text editor.
3. Select some text and try changing the font size and style.

   When a style is selected, the corresponding button is depressed; plain is indicated by the absence of all three styles. (If it doesn't behave like this, you forgot to set the ButtonType property to Toggles.)

   Since the code to update the style controls is in place, these buttons update as you move the insertion point between styled and unstyled text.

Figure 46. Bold and Underline styles applied to text.



4. When you are finished testing, quit out of the test application to return to the Development environment.

Implementing the Color Control

In this section, you will add a Color control that enables the user to add a color attribute to text. We will use a Canvas control to provide this functionality.

The Canvas control is a blank "canvas" that comes equipped with drawing tools that enable you to control its appearance. The drawing tools consist of the methods in the Graphics class. We will use these drawing tools to draw a black border around the control and 'paint' the interior of the control with the color of the selected text. We will also give the control an action—it will display the Color Picker when the user clicks on it. In other words, it will work like a pushbutton but we will use the methods of the Graphics class to control its appearance. Unlike a PushButton control, it will also display the selected color.

To add the Color control and its label, do this:

1 In the Window Editor, click the StaticText object that serves as a label for the Size menu and duplicate it (press Ctrl+D or ⌘-D on Macintosh).
Be sure you have quit out of the test application before returning to the IDE.

2 Drag the duplicated object to the right of the Underline button and align it with the baselines of the other controls using the horizontal alignment guide. (You may need to increase the width of the window to do this.) Change its Text property to **Color:**. Set its Left property to **377** and its Width property to **39**.

3 Click on the Canvas control in the Controls list to select it.

4 With the mouse pointer, draw a rectangle to the right of this StaticText and within the bounds of the header area, about the same size as its StaticText label.
Your rectangle should look like this.

Figure 47. Creating the Color control by dragging.

When you release the mouse button, REALbasic adds a Canvas control the size of the area you drew.

5 Use the Properties pane to assign it the following properties:

Table 5: Properties of the Canvas Control.

Property	Value
Name	ColorButton
Left	419
Top	4
Width	57
Height	16

The next series of steps draws a black border and fills the Canvas control with the default text color.

To create the default appearance, do this:

1 Double-click the ColorButton to open its Paint event in the Code Editor for TextWindow.

The Paint event runs whenever REALbasic determines that the Canvas control needs to be redrawn. It is the place to update its appearance each time the user clicks it.

Notice in the Sub statement for the Paint event that it is passed one parameter, g as Graphics. The Graphics class contains a number of methods that you can use to draw shapes inside the Canvas and set its border and interior colors. You use this parameter to gain access to the Graphics class's drawing tools. You use the "dot" notation to do this.

2 Enter the following code into the Paint event:

```
g.ForeColor=&c000000  //black
g.DrawRect(0,0,g.Width-1,g.Height-1)
g.ForeColor=TextField.SelTextColor
g.FillRect(1,1,g.Width-2,g.Height-2)
```

The "dot" notation indicates that each line accesses a method or property of the Graphics class. For example, the first line, "g.ForeColor", accesses the ForeColor property of the Graphics class.

The ForeColor property specifies the color used by subsequent calls to any Graphics class method that does any drawing. The following line assigns a color to the ForeColor property.

```
g.ForeColor=&c000000  //black
```

The &c symbol specifies a color value using the Red, Green, Blue (RGB) color model. The first two digits are the amount of Red, the next two, the amount of Green, and the last two. the amount of Blue. In the Code Editor, you can see that each pair of digits is color-coded. The amounts range from 0 to 255 (in base 10), but the amount is specified in hexadecimal (a.k.a., base 16). This means that the values range from 00 to FF. The complete absence of red, green, and blue results in black. So, &c000000 is black. The expression "&cFFFFFF" specifies the color white.

The first line of code sets the foreground color, ForeColor, to black and the next line draws a border around the control's edges using the current value of ForeColor. The four parameters are the top and left coordinates of the rectangle to be drawn and the width and height of the rectangle. It starts at the top-left corner (0,0) and uses the width and height of the parent control to get those values. The Width and Height properties return the current width and height of the drawing region. Its better to use Width and Height rather than putting specific values in the code. If the control is resized, you don't have to modify these lines of code.

The next two lines set the ForeColor property to current text color in TextField and then paints the interior of the Canvas control with that color.

The next step is to make the Canvas control behave like a pushbutton. You will use the MouseDown event handler of the Canvas control. For our purposes, it works like the Action event handler of the BevelButton control. It runs when the user presses the mouse within the Canvas control. You will notice that it returns the coordinates of the mouse press, but you don't need to use them.

3 Highlight the MouseDown event handler of ColorButton and add the following code to the event:

```
Dim c as Color
Dim b as Boolean
c=&cFFFFFF   //default color of white
b=SelectColor(c,"Select a Text Color")
If b then //if the user selected a color
  Me.Graphics.ForeColor=c
  Me.Graphics.FillRect(1,1,Me.Graphics.Width-2, Me.Graphics.Height-2)
  TextField.SelTextColor=c
End if
```

The SelectColor function displays the Color Picker. It takes two parameters, a color and a text string that is displayed within the Color Picker dialog. The color you pass to SelectColor controls the appearance of the color wheel when the dialog appears. SelectColor returns a Boolean value that is True if the user clicked OK and False if the user canceled out of the dialog box.

If the user clicks OK, the selected color is returned in the variable, c.

NOTE: The value of c returned from SelectColor is different from the value passed to it (provided the user selected another color). This is possible because the color parameter is passed by reference rather than by value. That is, a reference (or pointer) to the variable rather than the actual value of the variable was passed. The function is then able to change the value and return the reference. You can make use of passing parameters by reference in your own methods by using the ByRef keyword in REALbasic's language. See the Language Reference for more information about ByRef.

Updating the Color Control

By now you must have guessed that we need to add some code to the SelChange event of TextField to update the color of the Canvas control when the user moves the insertion point into text that has a different color attribute.

To add the update code, do this:

1 In the Code Editor browser, click on SelChange in the TextField item and add the following code to the existing code:

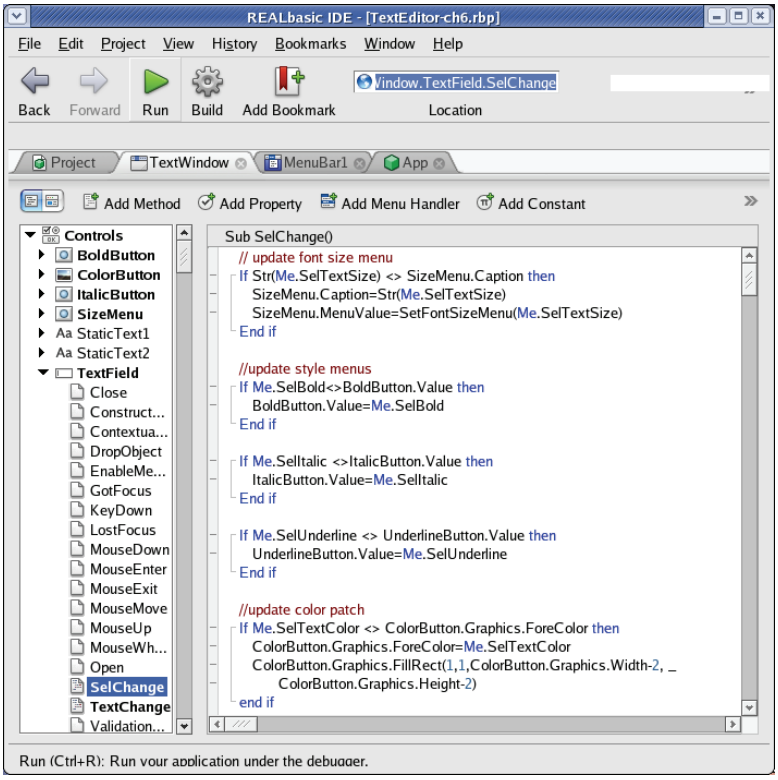
```
//update color patch
If Me.SelTextColor <> ColorButton.Graphics.ForeColor then
  ColorButton.Graphics.ForeColor=Me.SelTextColor
  ColorButton.Graphics.FillRect(1,1,ColorButton.Graphics.Width-2, _
                    ColorButton.Graphics.Height-2)
End if
```

Note that the second line of code inside the If statement is actually one logical line long but is too long to print on one line in the manual. Use the Underscore keyword

as the last character of the first line in the Code Editor — right after the comma — to continue the line on the next line.

The SelChange event handler should now look like Figure 48.

Figure 48. The SelChange Event Handler.



This code follows the previous logic: If the color of the selected text is different from the current ForeColor property of the Canvas control, we update the ForeColor property and use the FillRect method of the Graphics class to repaint the interior of the Canvas control.

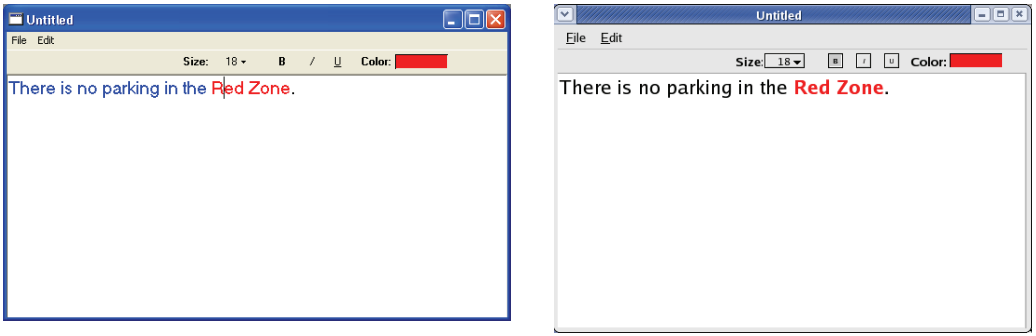
Testing the Color Control

Now that all the code is in place, you are ready to see how it works.

To use the styled text editor, do this:

1. Click the Run button in the Main toolbar and enter some text in the text editor.
2. Select some text and try changing the color.

Figure 49. Colors applied to text.



3 When you are finished testing, quit out of the test application to return to the REALbasic IDE.

Review

In this chapter you learned how to work with StaticText, BevelButton, and Canvas controls and used conditional compilation.

To Learn More About:	*Go to:*
REALbasic Controls	REALbasic User's Guide: Chapters 3, 5, 7.
REALbasic Standalone Applications	REALbasic User's Guide: Chapter 14.
REALbasic commands and language	REALbasic Language Reference

CHAPTER 7

Creating Dynamic Menus

In this chapter you will learn how to create a menu whose items will be created on-the-fly. You will add a Font menu to the application and add code that will load the names of the fonts installed on the user's computer.

Unlike the Size menu, you cannot specify the Font menu items in advance. Different users will see different Font menus.

Getting Started

If it is not already open, locate the REALbasic project file that you saved at the end of last chapter ("TextEditor-ch6"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch6" that is in the Tutorial Files folder on the REALbasic CD.

Implementing the Font Menu

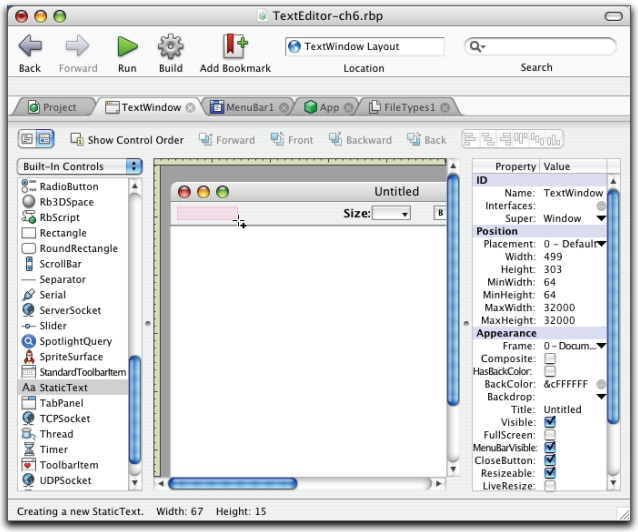
Implementing the Font menu involves the same basic steps for menu creation that you learned in the previous chapter. The key difference here is that you will add a method that loads the names of existing fonts into the menu items. This method runs when the window opens.

First, you add the Font menu to the header area of TextWindow.

To create the Font menu and its label, do this:

1 Open TextWindow's Window Editor and click on the StaticText control in the Controls pane. Create a rectangle by dragging on the left side of the toolbar, as shown in Figure 50.

Figure 50. Creating the StaticText control by dragging.



2 Using the Properties pane, change its properties as follows:

Table 6: Properties of the StaticText Control.

Property	Value
Left	7
Top	4
Width	38
Height	16
Text	Font:
TextAlign	Right
TextFont	System
TextSize	0
Bold	True (checked)

3 Select the BevelButton control in the Controls pane and draw one just to the right of the Font label and align it with the tops of the other controls.

4 Using the Properties pane, change its properties as follows.

Table 7: Properties of the Font Menu Control.

Property	Value
Name	FontMenu
Left	48
Top	4
Width	135
Height	16
Caption	(change to blank)
CaptionAlign	Center
HasMenu	Normal Menu
TextFont	System
TextSize	9

5 Save your project as **TextEditor-ch7**.

Building the Font Menu

We can build the items for the Font menu when the user opens a new instance of TextWindow. Therefore, we will add code to build the menu items to the Open event for FontMenu.

To build the Font menu items, do this:

1 Double-click the FontMenu control in TextWindow.
The Code Editor for TextWindow opens, with the Action event for FontMenu selected.

2 Select the Open event handler for FontMenu and add the following code to the method:

```
Dim nFonts as Integer
nFonts=FontCount-1

For i As Integer=0 to nFonts
  Me.AddRow Font(i)
Next

Me.Caption=TextField.SelTextFont
```

The FontCount function returns the number of fonts on the user's computer. The Font function is in a loop that runs from the first to the last font on your computer, with the local variable "i" keeping track of the number of fonts that have been added.

Notice that the declaration for the counter variable used by the For loop is inside the For statement. The expression:

```
For i As Integer=0 to nFonts
```

declares the local variable "i" as an Integer variable and immediately puts it to use in the For loop. In this way, you can declare the data type of a counter variable without using a Dim statement.

The expression:

```
Font(i)
```

returns the name of the i^{th} font, as the loop runs from 0 to nFonts.

The AddRow method of the BevelButton class adds a new item to the menu and takes one parameter, the text of the menu item. The For…Next loop executes this line of code repeatedly until all font names have been added. Since the menu items are numbered starting with zero, the loop goes from zero to FontCount-1 rather than 1 to FontCount.

The last line sets the default value of the Caption property of the Font Menu to the default font in TextField. This is the TextFont property of TextField.

Handling the Font Menu

The Font Menu needs to set the currently selected text to the font that the user chooses from the Font menu. This will be done using the SelTextFont property of the TextField. It also needs to add a check mark next to the name of that font.

To handle Font menu events, do this:

1. In the Code Editor for TextWindow, expand the Action event for FontMenu.
2. Add the following code:

```
Me.Caption=Me.List(Me.MenuValue)
TextField.SelTextFont=FontMenu.Caption
```

The MenuValue property is the *number* of the selected menu item and the List method returns the text of the menu item corresponding to the number passed to it. That is, the first line sets the Caption property (the text displayed by the pop-up) to the font that the user chooses.

The second line assigns the selected font (i.e., the Caption property) to the SelTextFont property of the TextField. SelTextFont is the font of the selected text. If no text is selected, any text that the user types is in the SelTextFont font.

Updating the Font Menu

The last step is to add some code to TextField that updates the font displayed by the Font menu when the user moves the insertion point to text in another font. Since moving the insertion point changes the text selection, we will use the SelChange event for TextField.

To update the Font menu, do this:

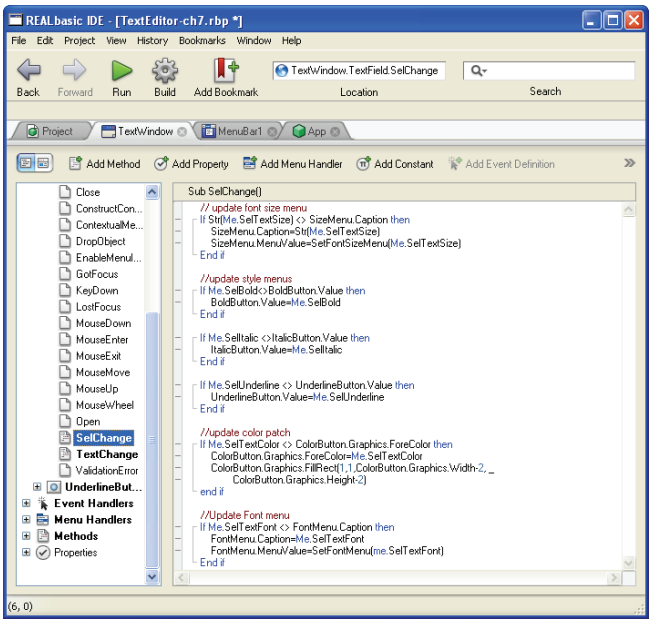
1 In the Code Editor for TextWindow, expand the TextField item and highlight the SelChange event.

2 Add the following code to this event handler:

```
//Update Font menu
If Me.SelTextFont <> FontMenu.Caption then
  FontMenu.Caption=Me.SelTextFont
  FontMenu.MenuValue=SetFontMenu(Me.SelTextFont)
End if
```

The SelChange method should now look like Figure 51 on page 81:

Figure 51. The SelChange method after adding code for the Font menu.



The If statement checks to see if the current font is different from what the Font menu indicates. If so, the next line resets the Caption property. The second line is needed to update the check mark that you see when the Font menu is pulled down. The MenuValue property is the number of the selected font, so we need to get the sequential number corresponding to this font. The SetFontMenu method does this. We need to add this to TextWindow to finish the job.

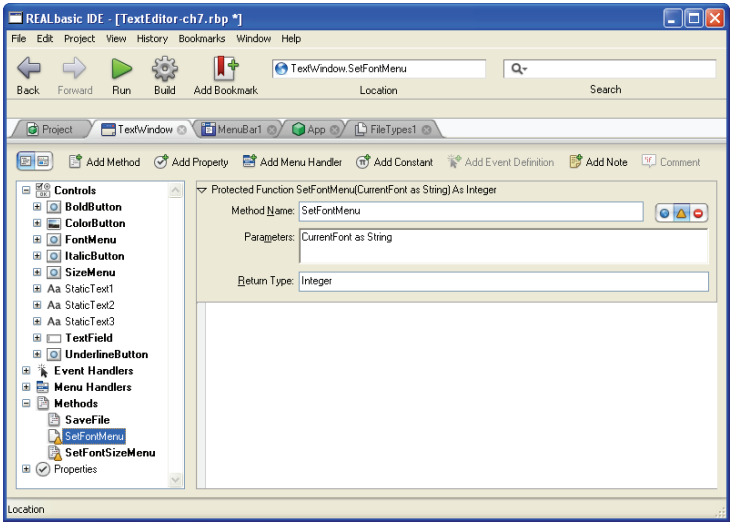
To add the SetFontMenu method, do this:

1 Click the Add Method button in TextWindow's Code Editor toolbar.
The Add Method declaration area appears above the code editing area.

2 Enter the name **SetFontMenu**, parameter **CurrentFont as String**, and Return type of **Integer**. Set the Access Scope to **Protected**.

The dialog box should now look like Figure 52 on page 82.

Figure 52. The SetFontMenu declaration.



Notice that the method is a function rather than a subroutine because you've specified a return type and the parameter, Font, is included in the function declaration line.

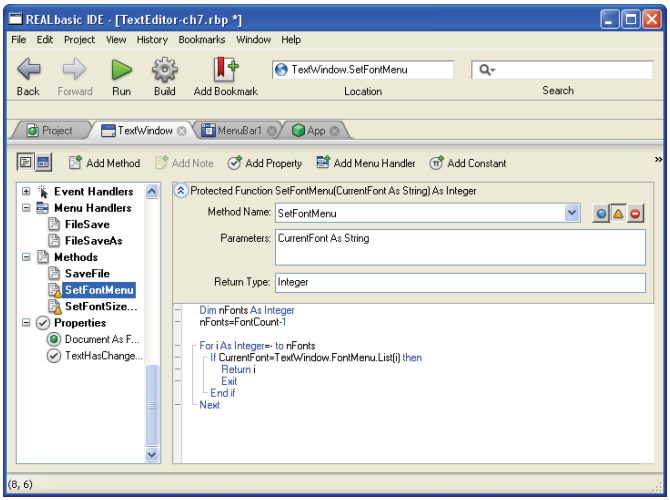
3 Add the following code to this function:

```
Dim nFonts as Integer
nFonts=Fontcount-1
For i As Integer=0 to nFonts
  If CurrentFont=textwindow.fontMenu.list(i) then
    Return i
    Exit
  End if
Next
```

4 Save the project.

The new method should look like Figure 53.

Figure 53. The SetFontMenu method.



In the SelChange event handler, the name of the current font is passed to this function in the parameter CurrentFont. The For…Next loop examines the name of each font on the user's computer until it finds the name of the current font. It then returns the sequential number, i, and aborts the loop using the keyword Exit.

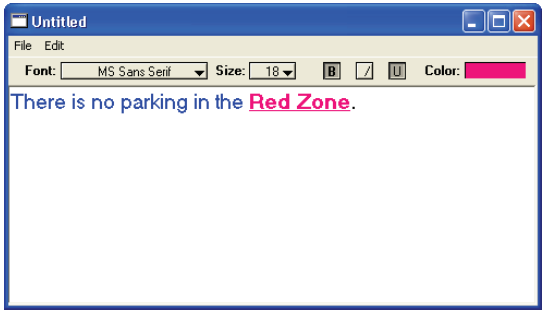
Once the SelChange event handler has this sequential number, it can assign it to the Font menu's MenuValue property. This resets the checkmark in the Font menu.

Testing the Application

Now that the entire header area has been built, you can test the application.

1 Click the Run button in the Main Toolbar and experiment with different fonts, font sizes, styles, and colors.

Figure 54. Colors, Fonts, and Styles on Windows.



2 When you are finished testing, quit out of the test application to return to the REALbasic IDE.

Review

In this chapter you learned how to dynamically create menu items in your application.

To Learn More About:	Go to:
REALbasic Font Handling	REALbasic User's Guide: Chapters 3, 4, 7.
REALbasic Controls	REALbasic User's Guide: Chapters 3, 5, 7.
REALbasic commands and language	REALbasic Language Reference

CHAPTER 8

Printing Styled Text

Now that you can change the font, font size, style, and color of text, you will want to be able to print out documents that retain your styled text attributes.

In this chapter, you will add Page Setup and Print items to the File menu to accomplish this task.

Getting Started

If it isn't already open, locate the REALbasic project file that you saved at the end of last chapter ("TextEditor-ch7"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch7" that is in the Tutorial Files folder on the REALbasic CD.

Creating the Page Setup and Print Menu Items

To create the menu items, do this:

1. Click the MenuBar1 tab to display the Menu Editor for MenuBar1 or, if it is not open, double-click the MenuBar1 item in the Project Editor.
2. Click on the File menu in the Menu Editor to display its menu items.
3. Click the Add Menu Item button in the Menu Editor toolbar.
   REALbasic adds a new, untitled menu item to the File menu.
4. In the Properties pane for the new menu item, enter **Page Setup...** in the Text area in the Appearance group and press Enter.
5. Deselect the AutoEnable property.
6. Click the Add Menu Item button in the Menu Editor toolbar again.
   REALbasic adds a new, untitled menu item to the File menu.
7. In the Properties pane for the new menu item, enter **Print...** in the Text area and press Enter.
8. Assign **P** to the Key property and select the MenuModifier property.
9. Deselect the AutoEnable property.
10. Drag the Page Setup... menu item below the Save As... menu item and then drag the Print... menu item below Page Setup....
11. Click the Add Separator button in the Menu Editor toolbar.
    REALbasic adds a separator between groups of menu items.
12. Drag the separator between the Save As and Page Setup menu items.
13. Click the Add Separator button again and drag it between the Print and Exit (Quit on Macintosh) menu items.
14. Save the project as **TextEditor-ch8**.

Enabling the Page Setup and Print Menu Items

You want the user to be able to access these menu items whenever a document window is open, so you should enable them in TextWindow's Code Editor. They need not be enabled when no document windows are open (this case occurs on Macintosh), so the AutoEnable property should not be used for these menu items. These two menu items are enabled whenever a document window is open.

To enable the menu items only when a document window is open, do this:

1. Click the TextWindow tab and open its Code Editor by clicking the Code Editor icon.
2. In the Code Editor for TextWindow, expand the Event Handlers s item in the browser area.

3 Highlight the EnableMenuItems event and add the following lines to the existing code:

```
FilePageSetup.Enable
FilePrint.Enable
```

Handling the Page Setup Menu Item

To store the user's selections from the Page Setup dialog box, you need to create a PrinterSetup object. This object has a property, SetupString, that contains many of these selections. You will first add this property to TextWindow's Code Editor.

On Linux, there is no Printer Setup dialog. Calling the PrinterSetup method returns False and the user is not shown any dialog box. For this reason, the menu handler for the Page Setup menu item uses conditional compilation to display a message to Linux users.

To add the property, do this:

1 In TextWindow's Code Editor, click the Add Property button in the Code Editor toolbar and enter **PageSetup as String** in the Property definition dialog box and click the Protected Access Scope button (the yellow triangle) if it is not already selected.

Protected access scope means that no objects outside TextWindow can read or set the value of this property. No objects outside the scope of the TextWindow need to access it, so this protects the property from possible programming errors in code outside TextWindow.

Next, you need to write the menu handler for the Page Setup menu item.

To add the Page Setup menu handler, do this:

1 Click the Add Menu Handler button and choose FilePageSetup from the pop-up menu.

2 Enter the following code in the Page Setup menu handler:

```
#if Not(TargetLinux) then //do not call PrinterSetup on Linux
   Dim ps as PrinterSetup
   ps=New PrinterSetup

   If PageSetup <> "" then
    ps.SetupString=PageSetup
   End if

   If ps.PageSetupDialog then
    PageSetup=ps.SetupString
   End if
#else
   MsgBox "Linux does not support the Printer Setup dialog!"
#endif
```

The PrinterSetup property, SetupString, contains the page setup selections that the user makes in the Page Setup dialog box. If settings are stored in the property PageSetup, they are assigned to the SetupString property.

The second If statement displays the Page Setup dialog box. If the user clicks OK, PageSetupDialog returns True and the SetupString is assigned to the PageSetup property.

The menu handler uses the PageSetup property to store the user's selections.

Handling the Print Menu Item

You use an object of type StyledTextPrinter to print styled text. It uses its DrawBlock property to "draw" the styled text on the page.

To add the Print menu handler, do this:

1. Click the Add Menu Handler button in TextWindow's Code Editor and choose FilePrint from the pop-up menu.
2. Enter the following code in the Print menu handler:

```
Dim stp as StyledTextPrinter
Dim g as Graphics
Dim ps as New PrinterSetup
Dim pageWidth, pageHeight as Integer
ps=new PrinterSetup

If PageSetup <> "" then  //PageSetup contains properties
 ps.setupString=PageSetup
 pageWidth=ps.Width-36
 pageHeight=ps.Height-36

// open Print dialog with Page Setup properties
 g=openPrinterDialog(ps)
else
 g=openPrinterDialog() //open dg w/o Page Setup properties
 pageWidth=72*7.5 //default width and height
 pageHeight=72*9
End if
If g <> Nil then //user didn't cancel Print dialog
 stp=TextField.StyledTextPrinter(g,pageWidth-48)
 Do Until stp.EOF
  stp.drawBlock 36,36,pageHeight-48
  if not stp.eof then //is there text remaining to print?
   g.NextPage
  end if
 Loop
End if
```

The menu handler uses the StyledTextPrinter method of the EditField class to create a StyledTextPrinter object ("stp"). If the user used the Page Setup dialog box to set properties, the PageSetup property is not null and its properties are used for printing. The Width and Height properties of the PrinterSetup object are the width and height of the entire printable area, as defined in the Page Setup dialog box. Typically, a styled text document uses additional left, right, top and bottom margins. Thus, small values are subtracted from the Width and Height properties. You may want to use different values to suit your page size and Page Setup choices.

If the user does not display the Page Setup dialog box, default values for the height and width of the printable area are used.

Since the TextField may contain more than one page of text, we must support multiple page printing. The Boolean property of a StyledTextPrinter object, EOF, (end-of-file) is False until there is no more text to print. The Do loop executes repeatedly until EOF is True. It contains a call to the drawBlock method, which prints a block of text on the page.

```
stp.drawBlock 36,36, pageHeight-48
```

The first two parameters give the location of the top-left corner of the block on the page. They are offsets from the top-left corner of the printable area on the page, as defined in your Page Setup.

The third parameter gives the height of the block (The width of the block is given by the PageWidth variable, which was passed as a parameter to the StyledTextPrinter method).

Just after you print a block of styled text, you need to determine whether there is still more text to print. If so, you need to use the NextPage method of the Graphics class to generate a new page. This is handled by the If statement within the Do loop.

In the example code, the parameters passed to drawBlock were chosen so that the margins look good on 8.5" x 11" paper. If your page size is different (i.e., you use A4 paper), you should modify the values of pageWidth, pageHeight, and the location of the top-left corner of the printable area to suit your paper.

Testing Styled Text Printing

Now that styled text printing has been installed in your application, you are ready to see how it works.

To print styled text, do this:

1. Click the Run button in the Main Toolbar and enter some text in the text editor. If you encounter errors, be sure you've added the properties and methods to the TextWindow's Code Editor.
2. Select some text and change the font size and style.
3. Use the Page Setup and Print menus to test styled text printing.

4 When you are finished testing, quit out of the test application to return to the REALbasic IDE.

Review

In this chapter you learned how to add the capability to print styled text from your application.

To Learn More About:	Go to:
REALbasic Text Handling	REALbasic User's Guide: Chapters 7, 8.
REALbasic commands and language	REALbasic Language Reference

CHAPTER 9

Communicating Between Windows

In this chapter you will work with window communication in REALbasic. You will learn how to:

- Add a Find and Replace dialog box to your application,
- Add code to your application to allow communication between the dialog box and the text editor.

The Find function that you will use searches from the first character of the document to the end of the text. It is not case-sensitive.

Getting Started

If it isn't already open, locate the REALbasic project file that you saved at the end of last chapter ("TextEditor-ch8"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch8" that is in the Tutorial Files folder on the REALbasic CD.

Implementing the Find and Replace Menu Items

By now you are familiar with the process of adding a menu item, enabling it, and adding a menu handler. The new feature in this chapter is that the dialog box that is displayed by the menu item must communicate with another window in the application.

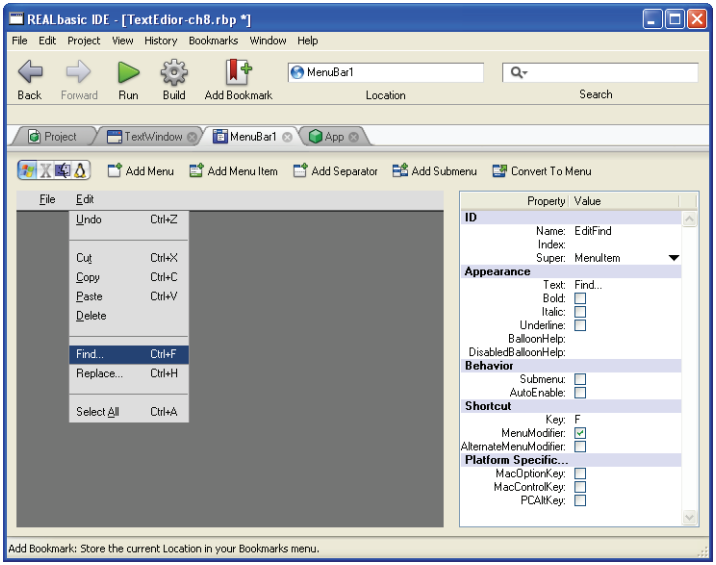
You will start by adding a menu item for the Find function to the Edit menu.

Creating the Menu Item

To create the menu item, do this:

1. Click the MenuBar1 tab in the Tab bar to display its Menu Editor or, if it is not open, double-click the MenuBar1 item in the Project Editor.
2. Click on the Edit menu in the Menu Editor.
   The Menu Editor displays the Edit menu's menu item.
3. Click the Add Menu Item button in the Menu Editor toolbar.
   REALbasic adds a new, untitled menu item to the Edit menu.
4. Use its Properties pane to change its Text property to **Find...**.
5. Type an **F** as the Key property and select the MenuModifier property.
6. Deselect the AutoEnable property.
7. Click the Add Menu Item button in the Menu Editor toolbar once again.
   REALbasic adds a new, untitled menu item to the Edit menu.
8. Enter **Replace...** as its Text property.
9. Type an **H** for the Key property and select the MenuModifier property.
10. Deselect the AutoEnable property.
    Drag the Find menu item above the Select All item and then drag the Replace menu item between the Find and Select All menu items.
11. Click the Add Separator button in the Menu Editor toolbar.
    This creates a separator between groups of menu items.
12. If it is not already between the Replace and Select All menu items, drag it there.
    The Edit menu should look like Figure 55.

Figure 55. The completed Edit menu.



13 Save your project as **TextEditor-ch9**.

Enabling the Find and Replace Menu Items

The Find and Replace menus should be enabled only when a document window is open, so you should enable it in TextWindow's Code Editor rather than the App object's Code Editor. Since the AutoEnable property was not selected, these menu items will be disabled when no document windows are open. This case can occur on Macintosh.

To enable the menu item, do this:

1 Open the Code Editor for TextWindow.
If the Window Editor is open, click its tab; if not, double-click its item in the Project Editor.

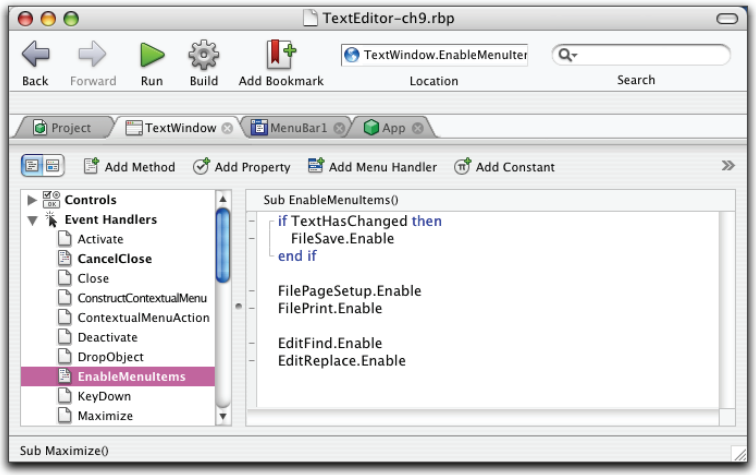
2 Open the Event Handlers group and highlight the EnableMenuItems event handler.

3 Add the following code to the end of the method:

```
EditFind.Enable
EditReplace.Enable
```

The Code Editor should look like that shown in Figure 56.

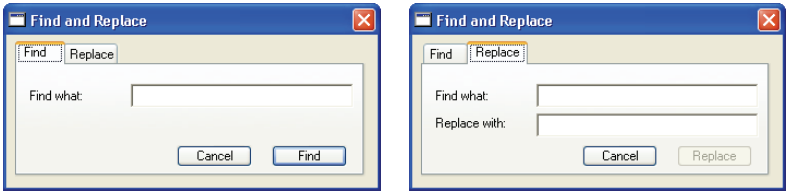
Figure 56. Updated Code Editor for EnableMenuItems.



Creating the Find and Replace Dialog Box

The next task is to create the dialog box itself. You are going to create one dialog box that works for both the Find and Replace functions. It will use a Tab Panel control that allows the user to select either Find or Find And Replace after opening the dialog box. When you are finished, the two tabs of the dialog box will look like Figure 57:

Figure 57. Find and Replace dialog box as it appears in a built application.



You begin by adding a new window to the project.

To create the dialog box, do this:

1. Click the Project tab in the Tab bar to display the Project Editor.
2. Click the Add Window button in the Editor toolbar.
   REALbasic adds a window to the project and names it Window1.
3. Double-click the Window1 item in the Project Editor to display its Window Editor.
4. Use Window1's Properties Window to change its name to **FindWindow**.
5. Change its Title property to **Find and Replace**.
6. Change the window's Width to **340** and Height to **140**.

7 Deselect the MenuBarVisible property.
This property was deselected because FindWindow will be a fixed-sized dialog box.

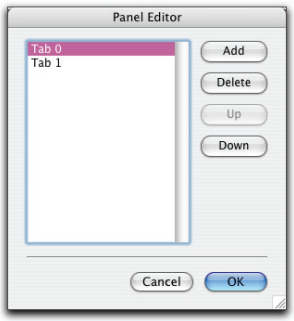
The following steps add the controls to the empty window.

1 Drag a Tab Panel control from the Controls pane to the FindWindow.

2 Move it to the top-left corner of the window and set its Left, Top, Width, and Height properties to **8**, **6**, **321**, and **124**.

3 In the TabPanel's Properties pane, click the Panels property, which reads "(Tabs)". The Tab Panel Editor dialog box appears. Use this dialog to create additional tabs and enter the label for each tab.

Figure 58. The Tab panel Editor.



4 Click on Tab 0 once to select it and then click again to get an insertion point.

5 Change its name to **Find**.

6 Click on Tab1 to select it and then click again to get an insertion point.

7 Change its name to **Replace**.

8 Click OK to put away the Tab Panel editor.
The TabPanel in FindWindow now has two tabs, with the labels Find and Replace.

The next series of steps adds the controls to the Find panel.

1 If it is not already selected, click the Find tab and drag a StaticText control to the top-left area of the TabPanel. This control will serve as the label for the entry area in the Find panel.

2 Set its Left, Top, Width, and Height properties to **21**, **48**, **84**, and **16** and Change its Text property to **Find what:**.

3 Drag an EditField control from the Controls pane to the right of the StaticText control and assign it the properties shown in Table 8 using the Properties Window.

Table 8: Properties of the EditField Control.

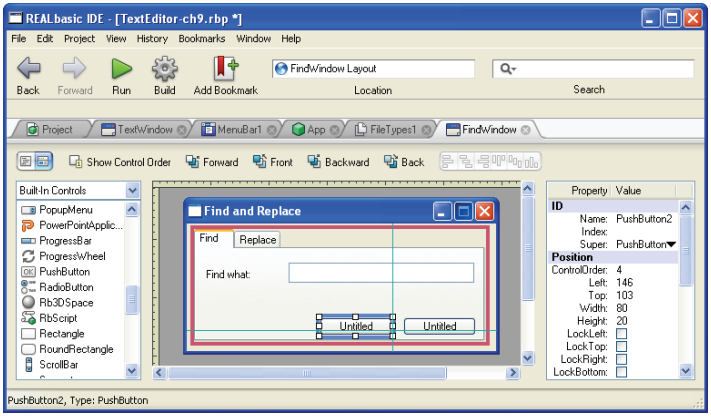
Property	Value
Name	FindText
Left	112
Top	43
Width	204
Height	22

4 Drag a PushButton from the Controls pane to the bottom-right corner of the TabPanel control, as shown in Figure 59, below.

5 Select the Pushbutton control and choose Edit ▶ Duplicate (Ctrl+D on Windows and Linux or ⌘-D on Macintosh) to create another pushbutton.

6 Drag the second pushbutton to the left of the first one, letting REALbasic align their baselines, as shown in Figure 59.

Figure 59. Aligning the Find button.



7 Select each pushbutton and make the property assignments shown in Table 9.

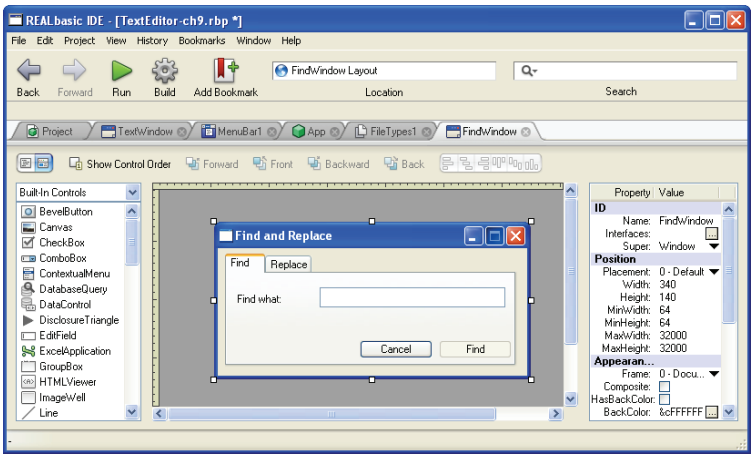
The pushbutton on the right is the Find button, as shown in Figure 60.

Table 9: Properties of the Cancel and Find Pushbuttons.

Property	Pushbutton	
	Cancel	Find
Name	CancelFind	FindButton
Left	156	243
Top	101	101
Caption	Cancel	Find
Default	Not checked	Checked
Cancel	Checked	Not checked
Enabled	Checked	Not Checked

The first panel of the dialog box should now look like this.

Figure 60. The Find panel of the Find and Replace dialog box.



Notice that the Find button is initially disabled, as it won't be enabled until the user enters the text to search for.

Next, you need to create the controls for the Replace Panel. You need to place four new controls in the exact positions occupied by the four controls on the Find page and add two controls for the Replace label and entry area.

1. Click on the Replace tab of the Tab Panel control.
   This hides the controls on the Find panel and allows you to place a new set of controls that will be shown only when the user clicks the Replace tab.
2. Drag a StaticText control to the top-left area. This control will serve as the label for the Find entry area on the Replace panel.
3. Set its Left, Top, Width, and Height properties to **21**, **48**, **84**, and **16**. and Change its Text property to **Find what:**.

4. Drag an EditField control from the Controls pane to the right of the StaticText control.
5. Click on the EditField control and, using the Properties Window, assign it the properties shown in Table 10.

Table 10: Properties of the EditField Control.

Property	Value
Name	SearchText
Left	112
Top	43
Width	204
Height	22

6. Duplicate the StaticText and Editfield Tools (Ctrl+D on Windows and Linux or ⌘-D on Macintosh) and move them into the approximate positions for the 'replace' controls.
7. Set the Left, Top, Width, and Height properties of the new StaticText control to **21**, **73**, **84**, and **16** and change its Text property to **Replace with:**.
8. Set the properties of the EditField control as follows.

Table 11: Properties of the EditField Control.

Property	Value
Name	ReplaceText
Left	112
Top	70
Width	204
Height	22

9. Next, drag a PushButton control from the Controls pane to the lower-right corner of the TabPanel control.
10. Select the Pushbutton control and choose Edit ▶ Duplicate (Ctrl+D on Windows and Linux or ⌘-D on Macintosh) to create the Cancel pushbutton.
11. Drag the Cancel pushbutton into place, to the left of the Replace button, letting REALbasic align it with the baseline of the Replace button using the alignment line.

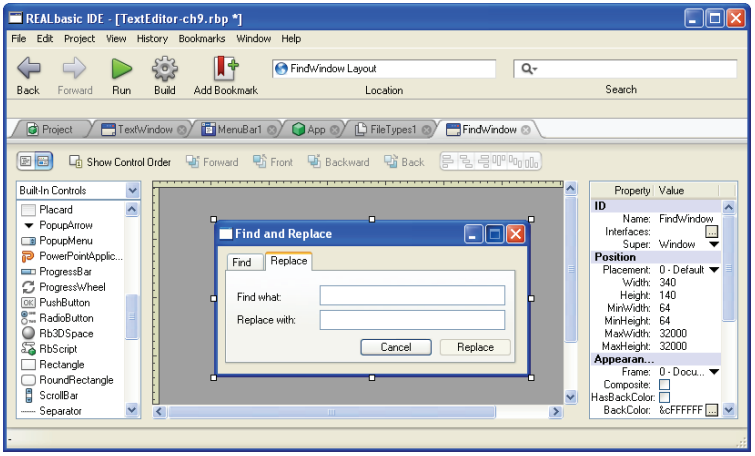
12 Select each pushbutton and make the property assignments shown in Table 12.

Table 12: Properties of the Cancel and Find PushButtons.

Property	Pushbutton	
	Cancel	Replace
Name	CancelReplace	ReplaceButton
Left	156	243
Top	101	101
Caption	Cancel	Replace
Default	Not checked	Checked
Cancel	Checked	Not Checked
Enabled	Checked	Not Checked

The second page of the Find and Replace dialog box should now look like this:

Figure 61. The Replace panel of the Find and Replace dialog box.



Specifying the Actions of each Control

Now that the dialog and menu items are built, you need to specify the actions of each control.

To specify each button's actions, do this:

1 Click the Code Editor icon in FindWindow's Window Editor to switch to its Code Editor.

You can also Control-click or Right+click anywhere in FindWindow's Window Editor to display its contextual menu and choose Edit Code. If you Right+click (Control-click on Macintosh) on a control, you can navigate directly to one of the control's event handlers.

2 Expand the Controls item.

You will see the names of the objects that you just placed in FindWindow.

3 Expand FindText (the entry area on the Find panel) and then click the TextChange event handler. It runs whenever a user enters or edits text in the Find panel of the dialog box. Enter the following code.

```
If Len(Me.Text) > 0 then //if the user entered text
  FindButton.Enabled=True
Else
  FindButton.Enabled=False
End if
```

The If statement determines whether the FindText field contains some text after the change (The Me function is a reference to the control that owns the event handler—in this case FindText). If so, it enables the Find button.

4 Expand SearchText (the 'Find' entry area on the Replace panel) and click the TextChange event handler. Add the following code to this event handler.

```
If Len(Me.Text) > 0 then
  ReplaceButton.Enabled=True
Else
  ReplaceButton.Enabled=False
End if
```

This code enables the Replace button on the Replace screen.

5 Expand CancelFind and then click Action. Then enter the following code:

```
Close
```

This line of code closes the window by calling the Close method of the Window class.

6 Expand CancelReplace and add the same line of code, **Close**, to its Action event handler.

7 Expand FindButton and then click Action. Then enter the following code:

```
TextWindow(Window(1)).Find FindText.Text,""
Close
```

This method uses a method called Find which does the real work. It takes as its parameters the text the user has entered into the Find panel of the dialog box, FindText.Text—the Text property of the EditField, FindText. If the user is doing a find and replace, the second parameter is the replacement string. In the case of a Find, we can pass an empty text string.

The Window function is used to specify the TextWindow in which to search. The expression "Window(1)" refers to the second window—Window (0) is the Find and Replace dialog itself—so Window(1) is the frontmost document window.

8 Expand ReplaceButton and highlight its Action event handler. Enter the following code:

```
TextWindow(Window(1)).Find SearchText.text,ReplaceText.Text
Close
```

This code passes the contents of ReplaceText to the Find method as the second parameter.

Adding the Find Method to TextWindow

The next step is to add the Find method to TextWindow. This method must be added to TextWindow rather than FindWindow because it runs when a user has a document window open.

To add the Find method, do this:

1 Click TextWindow's Tab in the Tab bar or, if it is closed, click the Project tab and then double-click the TextWindow item in the Project Editor.

2 If it is not displayed, switch to TextWindow's Code Editor by clicking the Code Editor icon in its Editor Toolbar (Ctrl+E or Command-E on Macintosh).

3 Click the Add Method button to create the Find method.

Remember to add this method to TextWindow's Code Editor, not FindWindow's.

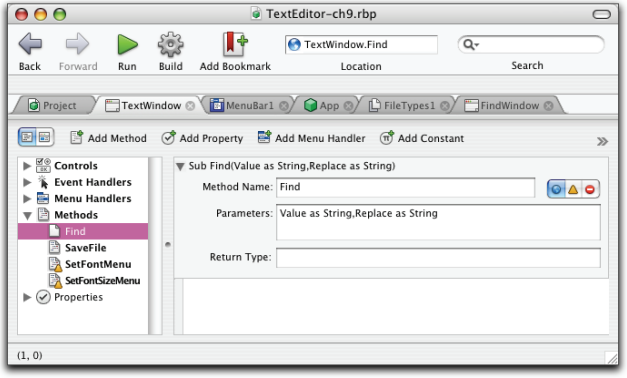
4 Enter **Find** as the method name and **Value as String,Replace as String** as the parameters.

5 Click the **Public** Access Scope button (the "globe" icon on the left).

This method needs to be Public because it will be called from outside TextWindow—from FindWindow.

The Find method's declaration area should look like that shown in Figure 62:

Figure 62. The method declaration for Find.

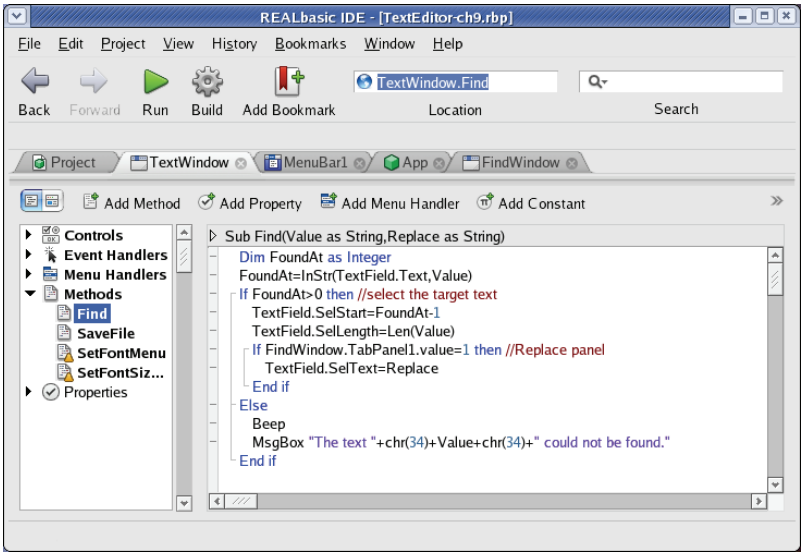


6 Enter the following into the Find Code Editor.

```
Dim FoundAt as Integer
FoundAt=InStr(TextField.Text,Value)
If FoundAt>0 then //select the target text
  TextField.SelStart=FoundAt-1
  TextField.SelLength=Len(Value)
  If FindWindow.TabPanel1.value=1 then //Replace panel
    TextField.SelText=Replace
  End if
Else
  Beep
  MsgBox "The text "+chr(34)+Value+chr(34)+" could not be found."
End if
```

The Code Editor should look like Figure 63.

Figure 63. The Find method in the Code Editor.



This method locates the string to be searched for (the parameter *Value*) using the built-in InStr function. InStr takes two parameters, the text to search and the text to search for within the text being searched. It then sets the SelStart property of TextField to the position of the first highlighted character and SelLength, the length of the highlighted text, is set to the length of the string to be searched for.

The second If statement checks to see if the user was using the Replace page of the dialog box (the Value property of a Tab Panel control returns the number of the current page, with the first page being page zero.) If it is, it assigns the second

parameter to the SelText property of TextField — making the replacement string the selected text.

The next step is to add the menu handlers for the Find and Replace menu items. The menu handler displays the correct tab panel in the dialog box.

To add the menu handlers, do this:

1. With TextWindow's Code Editor displayed, click the Add Menu Handler button.
2. Choose EditFind from the Menu Item pop-up menu and click OK.
3. Enter the following code into the menu handler method.

```
FindWindow.Show
```

"Show" is a method of the Window class. This line of code displays the dialog box.

4. Click the Add Menu Handler button again and choose EditReplace the Menu Item pop-up menu in the declaration area.
5. Add the following code to the Replace menu item's menu handler:

```
FindWindow.TabPanel1.Value=1  //display Replace panel
FindWindow.Show
```

This menu handler also shows the dialog box. The second line selects the second panel of the TabPanel control (the first panel is numbered zero) and displays it.

Testing the Find and Replace Functions

You are now ready to test the new features. Click the Run button in the Main toolbar, enter some text, and test the Find and Replace menu item.

Once you open the dialog, you can change your mind and display the other panel simply by clicking a tab.

You might uncover the following weakness yourself. If you enter text into either Find entry area and then switch panels, you'll notice that your text doesn't appear in the other panel's Find field. You can fix that easily.

To copy text to the other panel, do this:

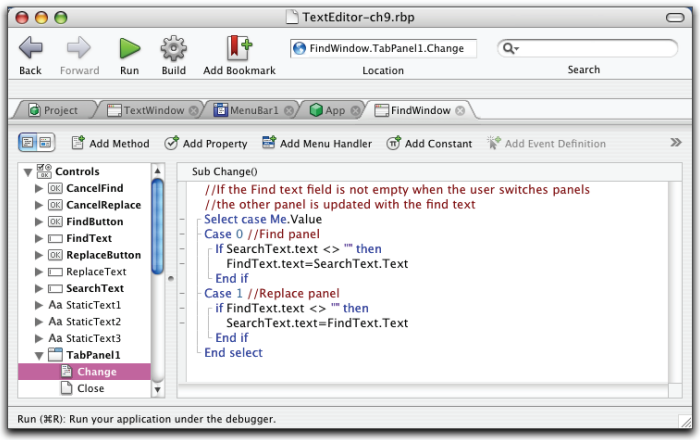
1. Return to the REALbasic IDE and click FindWindow's tab.
2. Click the Code Editor icon to switch the view to the Code Editor if it is not already displayed.
3. Expand the TabPanel1 item in the Controls group and highlight the Change event handler.
   This is the event that runs when the user clicks on a tab.

4 Enter the following code:

```
//If the Find text field is not empty when the user switches panels
//the other panel is updated with the find text
Select case Me.Value
Case 0 //Find panel
  If SearchText.text <> "" then
    FindText.text=SearchText.Text
  End if
Case 1 //Replace panel
  if FindText.text <> "" then
    SearchText.text=FindText.Text
  End if
End select
```

The Select Case statement takes the value of the TabPanel's Value property (which is the number of the panel that is displayed), and then copies the contents of the other panel's Find entry area into its Find entry area. The Code Editor for FindWindow should now look like this.

Figure 64. The Change event handler for the TabPanel control.



5 Save your project.

Try the application again. The Find and Replace dialog no longer loses what you have entered when you display the other panel.

Review

In this chapter you learned about the TabPanel control and how to create objects that communicate with each other in your application.

To Learn More About:	Go to:
REALbasic Controls	REALbasic User's Guide: Chapters 3, 5, 7.
REALbasic Object Communication	REALbasic User's Guide: Chapters 3, 5, 9.
REALbasic commands and language	REALbasic Language Reference

CHAPTER 10

Handling Errors in your Code

In this chapter you will work with the REALbasic Debugger and build a stand-alone application from your project. You will learn how to:

- Identify and fix syntax errors,
- Use the Debugger to find logical errors in your code,
- Handle runtime errors.

Getting Started

If your TextEditor project isn't already open, locate the REALbasic project file that you saved at the end of last chapter ("TextEditor-ch9"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch9" that is in the Tutorial Files folder on the REALbasic CD.

Using the Debugger

The REALbasic Debugger is the part of REALbasic that helps you fix parts of your application that aren't working properly. As with the rest of REALbasic, the Debugger is easy to use.

Automatic Debugging Features

While you are writing your code, you can take advantage of several features of the IDE that help reduce errors. The syntax coloring and code indentation in the Code Editor is one way that REALbasic proactively helps you to debug your code. Another is automatic syntax checking. Whenever you try to run your application, REALbasic checks the syntax of all your code and stops when it finds a syntax error.

To demonstrate REALbasic's syntax checking, do this:

1. Open the Code Editor for TextWindow and open the Methods item.
2. Select the SetFontSizeMenu method to display its code.
3. Change the line

```
s=0
```

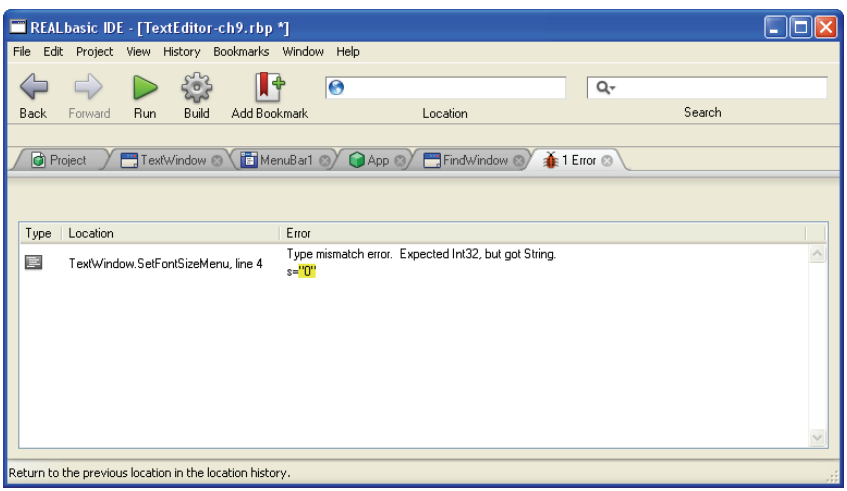
to

```
s="0"
```

This changes the data type of zero from a number to a string.

4. Now, click the Run button in the Main toolbar (Ctrl+R or ⌘-R).
A "Type Mismatch" error message appears in the Errors panel. The expression that produced the error is highlighted. The Errors panel should look like this.

Figure 65. A Syntax error message in the Code Editor.



The variable s was declared as an integer, so all the values you assign to it must be numbers. The Type Mismatch Error occurs when a value is an incorrect data type. You can double-click on the error to go to the method to correct the error.

5 To fix the error, double-click the error message in the Errors panel and delete the quotes from the line of code.

6 Retest the application.

Now that there are no syntax errors, REALbasic is able to compile your code. The bug does not disappear until you recompile the project.

Using the Debugger to Find Logical Errors

Errors that occur while your program is running are usually logical errors. To debug these errors, you will need to indicate to the REALbasic Debugger where it should check your code.

First, you need to set *breakpoints* in the source code in the region where you think the program is failing. Breakpoints are locations in your code where the application will pause and enter the Debugger while it is running. Once you are in the Debugger, you can examine the current values of variables, properties, and other parameters. You can check for unexpected, improper, or undefined values and take appropriate corrective action. You can also verify that your methods are actually being called when you expect them to be called.

Breakpoints don't alter your code and do not pause a stand-alone application built with REALbasic. The following exercise shows you how you can pause the application, check on the current values of variables, and continue executing a method line-by-line.

To see how the REALbasic Debugger works, do this:

1 If it is not already open, switch to the Code Editor for TextWindow.

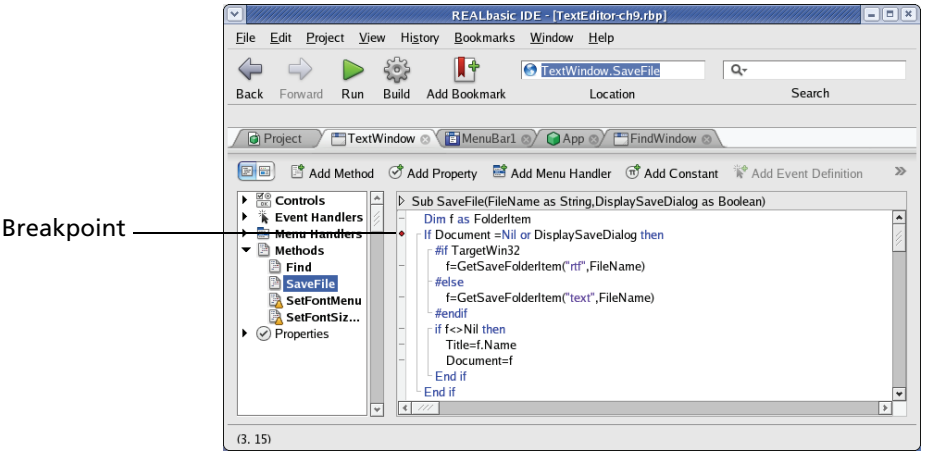
2 In the Browser, expand the Methods item and select the SaveFile method.

The dashes in the first column indicate where you can set breakpoints.

3 Click on the dash in the line containing the first "If" keyword to set a breakpoint.

A red circle appears in the margin of the Code Editor, signaling a breakpoint. The Code Editor should look as shown in Figure 66 on page 110.

Figure 66. Setting a Breakpoint in the Code Editor.



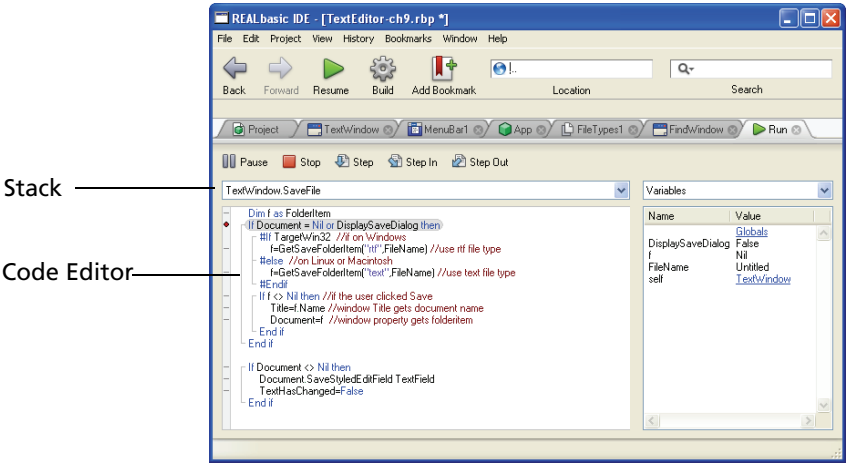
This breakpoint will cause REALbasic to pause when it tries to execute this line. This happens you try to save a document. When you try to save a new document, the Debugger will appear instead of the save-file dialog box.

4 Click the Run button in the Main toolbar (Ctrl+R or ⌘-R) to test your application in the Debugger.

5 Type some text into the text editor and choose File ▶ Save (Ctrl+S or ⌘-S).

This menu command calls the SaveFile method. It runs until it gets to the line of code at which you have placed the breakpoint. When it reaches the breakpoint, it stops and displays the REALbasic Debugger in the Run panel of the IDE.

Figure 67. The Debugger stopped at the breakpoint.

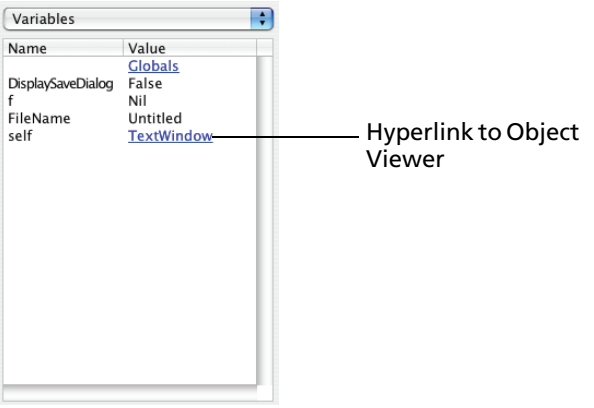


The Debugger screen is divided into three sections. The Code Editor section shows the method that is currently executing. Execution has stopped right at the breakpoint line, which is indicated by the red dot. The highlighted line of code indicates the line of code that is executing.

The Stack popup menu contains the name of the current method, along with any methods that invoked the current method. They are listed in the order that they were called. You can check the Stack pop-up menu to verify that methods are actually called when you expect them to be called.

Variables Pane contains a list of all the variables local to the method containing the breakpoint, along with their current values (if any).

Figure 68. The Variables pane.



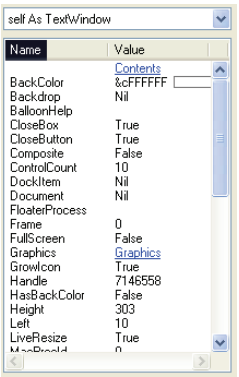
In this instance, the variable DisplaySaveDialog is the parameter passed to the method. Its current value of False is shown. The value of the String parameter FileName is “Untitled”.

Any objects (rather than variables) that are defined in the method are shown as hyperlinks rather than values. In this example, the object, f, is a FolderItem that will be shown as an object but its value is currently undefined (“Nil”). It has been declared but has not yet received a value.

The variable Self refers to the window itself and it is of type TextWindow. It is defined and therefore its value is represented by a hyperlink to an Object Viewer.

If you click a link, the Variables pane changes to show the variables for the object you clicked on.

Figure 69. The Object Viewer for TextWindow.

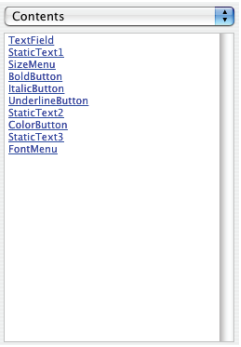


Each property of the object is shown in the same format as the Variables Pane. If a value is also an object, then a hyperlink is shown instead of a value. For example, the Graphics property is shown as a hyperlink to its variables pane.

To return to the Variables pane for this method, choose it from the pop-up menu above the list.

The Variables pane for a window has a hyperlink called “Contents”. It links to a pane that lists all the controls in the window. Since every control is an object, each is shown as a hyperlink.

Figure 70. The Contents pane for TextWindow.



The important feature of the Variables pane is that it is interactive. As you execute code line by line in the Debugger, the Object Viewers and Variables pane update values in real time. In this way, you can see whether a particular value (or lack of a value) is causing a problem.

Using the Debugger’s Toolbar, you can control execution. Instead of just allowing your code to run indefinitely, you can control execution on a line-by-line basis.

The Debugger’s Toolbar has five buttons that perform the functions of the Debug menu items:

- **Pause**: Used to interrupt the application immediately.
- **Stop**: Stops execution and returns to the REALbasic IDE. This also exits from the Debugging environment, but without executing any more code.
- **Step Over**: Executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger executes the method but will *not* step through the method's code.
- **Step Into**: Executes the current line and moves on to the next line. If the current line includes one of your methods, the Debugger displays the method and steps through the method's code.
- **Step Out**: Executes the rest of the method without stopping on each line. This is handy when you have used Step Into to step through a method that was called by another method and now wish to continue code execution without stopping on each line.

  Also, the Run button in the Main toolbar has changed to **Resume**. Clicking it will cause the application to continue running from the current line of code.

  In the Variables Pane you see that the variable f is undefined. This is as it should be since the document that you are trying to save has not yet been saved. The variable f will be defined when you actually save the document.

  When you are in the Debugger, you can execute code line by line and monitor the contents of the Variables Window. You do this using the Step Into or Step Over buttons (or menu items).

6. Click the Step button until the save-file dialog box appears.
   Each time you select this menu item, the current line of code is executed and the highlight moves down one line.
7. Save the document under a filename and then examine the Variables Pane.
   Notice that the value of the f variable has changed from Nil to FolderItem because it has just been defined.

   Click the FolderItem hyperlink to see the current value of f.
   Notice that the AbsolutePath property now has a pathname to the document and the filename that you just gave it.

Figure 71. The Variables pane for the FolderItem after saving the document.

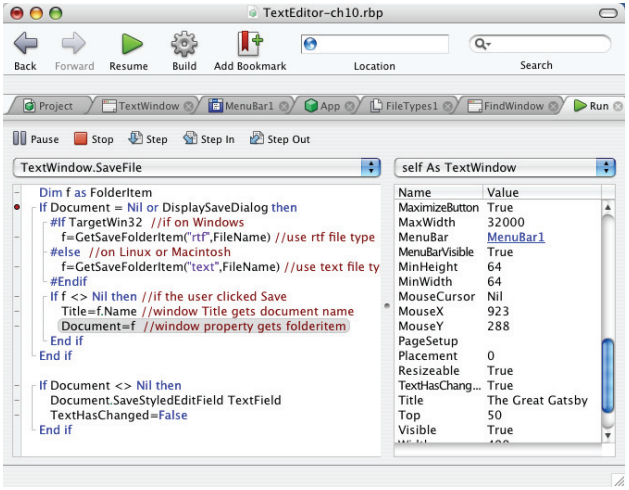
8 In the Variables pane, click the TextWindow object to see the parent window's properties and resize the Object Viewer so that you can see the Title property.

When you execute the line "Title=f.Name", notice that the Title property in TextWindow's Variables pane updates to show the document name that you entered.

9 Click the Step button again to execute this line.

You will see the value of the Title property change from "Untitled" to whatever filename you entered into the Save File dialog box.

Figure 72. Title property gets Name, shown in Figure 71.



10 To resume execution of your application, click the Resume button in the Main toolbar.

The debugging window disappears and the test application picks up where it left off. The name of the saved document is now in the title bar.

11 Choose My Application (Mac OS X) ▶ Quit on Mac OS X or File ▶ Exit (on Windows) to exit the Debugging environment and return to the Development environment.

Please refer to the *User's Guide* for a complete description of REALbasic's debugger.

Handling Runtime Errors

Sometimes you will find errors in your code that manifest themselves only when the line of code actually executes. These errors are called *runtime errors* because they occur at runtime rather than during syntax checking. Unless you handle runtime errors, a standalone version of your application will crash when the line of code containing the error executes.

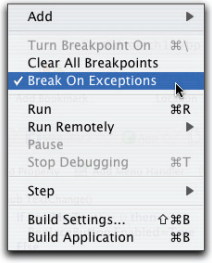
The existence of a potential runtime error does not prevent REALbasic from successfully compiling the application and the application may run without problems for a long while as long as the line of code containing the error is not actually executed. For example, if the line containing the error is in an If statement and the condition that would cause the line to execute is never True, the application will run normally.

Runtime errors can be handled by the Break on Exceptions option in the Project menu. When this option is selected, REALbasic will stop at the runtime exception as if you had set a breakpoint at that line. The Break on Exceptions option is selected by default.

To create a sample runtime error, do this:

1 Check that the Break on Exceptions options is selected in the Project menu. It should have a check mark beside it.

Figure 73. The Break on Exceptions option is selected.



2 Expand the Controls item in TextWindow's Code Editor and open the FontMenu's event handler.

3 Highlight FontMenu's Open event.

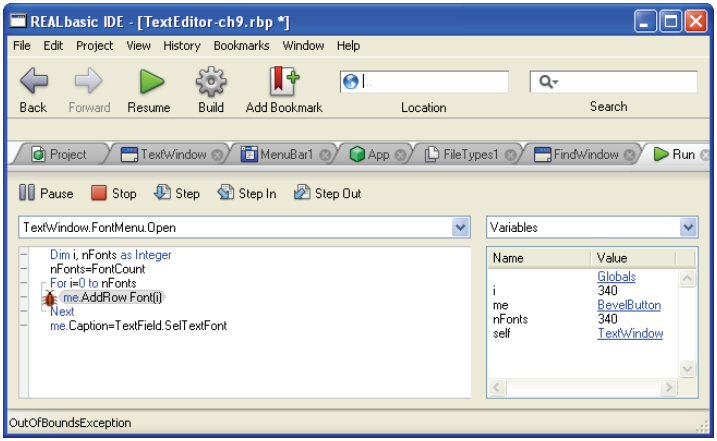
You will see the code:

```
Dim i, nFonts as Integer
nFonts=FontCount-1
For i=0 to nFonts
  me.AddRow Font(i)
Next
me.Caption=TextField.SelTextFont
```

4 Change the definition of nFonts from **FontCount-1** to **FontCount**.

5 Click the Run button in the Main toolbar. (Ctrl+R or ⌘-R).
Instead of seeing a new document window, execution will stop and you will see the error shown in Figure 74.

Figure 74. An Unhandled Runtime Exception Error.



What has happened is that the value of *i* in the For loop has reached the value of FontCount. Since the first menu item is numbered zero rather than one, this value forces the code to try to add one more item to the Font menu than there are fonts on the user's system. The value of *i* is now out of bounds. There is no syntax error, but it is a runtime error.

The bug appears in the line of code that contains the exception and the Tips bar contains the name of the exception.

When a runtime exception occurs in a standalone application, REALbasic can't display the line of code that caused the error. In a standalone application, all of the source code that you work with has been compiled into machine language code. Instead it displays a generic dialog box. The application has to shut down because it doesn't know what to do with the value that is out of range.

6 Click the Edit Code button in the Run toolbar and correct the deliberate error in the line:

```
nFonts=FontCount
```

by changing it back to:

```
nFonts=FontCount-1
```

7 Save your project as **TextEditor-ch10**.

Now, the application will run without errors.

Review

In this chapter you learned about syntax error messages, how to use the REALbasic Debugger, and how to handle runtime errors using the Break on Exceptions menu item.

To Learn More About:	*Go to:*
REALbasic Debugger	REALbasic User's Guide: Chapter 10.
REALbasic commands and language	REALbasic Language Reference

CHAPTER 11

Building a Standalone Application

In this chapter you will build a stand-alone application from your project. You will learn how to:

- Turn your project into stand-alone Windows, Macintosh, and Linux applications.

  The Professional version of REALbasic includes a cross-compiler that can build stand-alone applications for all three platforms. The Standard version of REALbasic builds stand-alone applications for the platform on which REALbasic is running and demo versions of the application for the other platforms. A demo version runs for only five minutes and then quits automatically.

Getting Started

If your TextEditor project isn't already open, locate the REALbasic project file that you saved at the end of last chapter ("TextEditor-ch10"). Launch REALbasic and open the project file. If you need to, you can use the file "TextEditor-ch10" that is in the Tutorial Files folder on the REALbasic CD.

Working with the Build Settings Dialog Box

If you have tested your project and everything works as expected, then you will want to turn your REALbasic project into a stand-alone application. As a stand-alone application, your program will work like any other Windows, Macintosh, or Linux application.

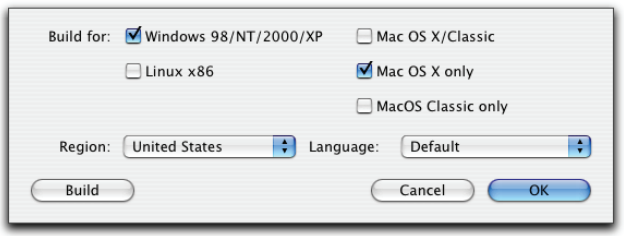
NOTE: Once you build a stand-alone version of your REALbasic application, you do not need to have REALbasic to run the application.

You use the Build Settings menu item in the Project menu to tell REALbasic which platform to build for. You can build for as many target platforms a you wish simultaneously. However, if you don't own the Professional version of REALbasic, only the version for your current operating system will be a full version.

To create a stand-alone application from your REALbasic project, do this:

1 Choose Project ▶ Build Settings….
The dialog box shown in Figure 75 appears.

Figure 75. The Build Settings dialog box.



In the top area of the dialog box, you select the target platform (or platforms) for the build. Your choices are Windows (Windows 98 to NT/2000/XP), Mac OS X or "classic" (PEF format), Mac OS X only (MachO format), Mac OS "classic" only, and Linux for x86 computers with GTK+ version 2 or above installed. The platform on which REALbasic is running is selected by default.

2 If you have other operating systems available, check them as well and click OK to close the dialog.

Working with the Application's Properties

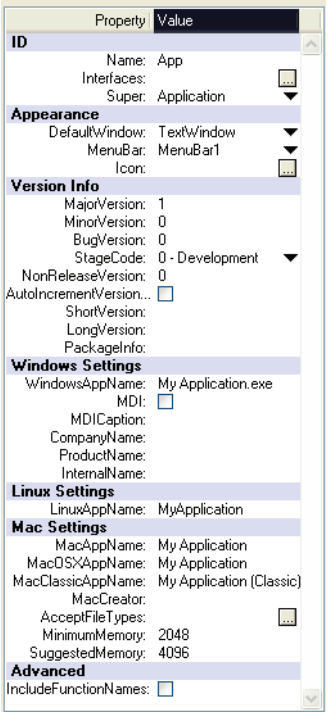
You can set many other properties of the built application with the App object's properties. You can set version information, platform-specific settings for each platform, and an advanced setting related to debugging. Of these, you will undoubtedly want to change the default name of the built application from "MyApplication".

To set the application's properties, do this:

1 Click on the App class object in the Project window.

The Properties pane changes to show the App object's properties. In the Windows Settings, Linux Settings, and Mac Settings groups you will see fields for the name of the built application for that operating system.

Figure 76. The App class's properties.



2. Change the name or names of the applications you are building to **TextEditor.exe** for Windows or **TextEditor** for Linux and Macintosh. If you are building for both Macintosh Classic and Mac OS X, add **(Classic)** to the name of the Classic build.

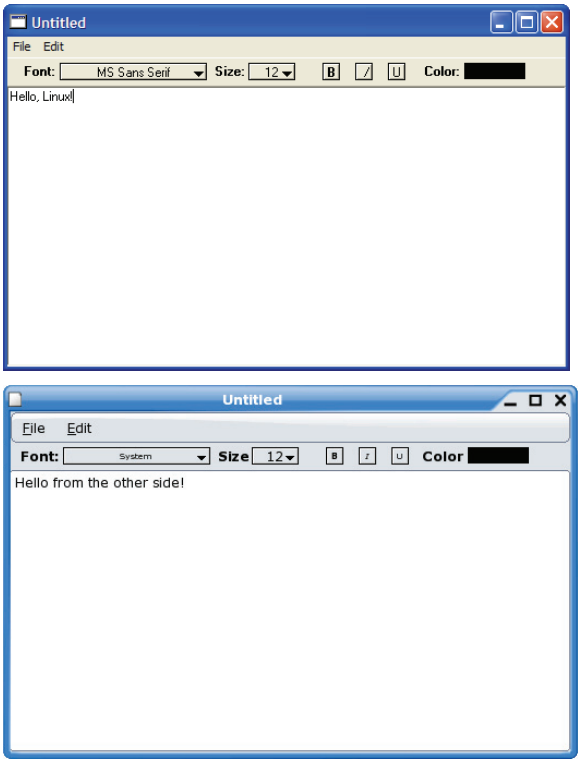
3. Click the Build button in the Main toolbar, not the Run button.

   REALbasic builds the standalone applications that you requested in the Build Settings dialog, places them in the same folder as your project, and brings that window to the front.

   You can now quit REALbasic and double-click the TextEditor icon from the desktop to edit text to your heart's content.

   If you built a standalone application for another platform, try out the standalone application under that platform as well.

Figure 77. TextEditor running on Windows and Linux.



NOTE: To learn about the other options in the Build Application dialog, consult the REALbasic User's Guide.

Review

In this chapter you learned how to build a stand-alone application from your REALbasic project.

To Learn More About:	*Go to:*
Building stand-alone Applications	REALbasic User's Guide: Chapter 13.
REALbasic commands and language	REALbasic Language Reference

Index